Stable scripting runtime: known limitations

Leave feedback

Unity supports a modern .NET runtime. You may encounter the following issues when using the .NET runtime:

Code size

The stable scripting runtime comes with a larger .NET class library API than the legacy scripting runtime. This means the code size is frequently larger. This size increase may be significant, especially on size-constrained and Ahead-of-Time (AOT) platforms.

To mitigate code size increases:

Choose the smallest .NET profile possible (see .NET profile support). The .NET Standard 2.0 profile is about half the size of the .NET 4.x profile, so use the .NET Standard 2.0 profile where possible.
Enable **Strip Engine Code** in the Unity Editor Player Settings (go to **Edit** > **Project Settings** > **Player**). This option statically analyzes the managed code in the Project, and removes any unused code. **Note**: This option is only available with the IL2CPP scripting backend.

2018–03–15 Page amended with editorial review

Generic Functions

Leave feedback

The Unity Scripting API Reference documentation lists some functions (for example, the various `GetComponent` functions) with a variant that has a letter `T` or a type name in angle brackets after the function name:

```
//C#
void FuncName<T>();
```

These are generic functions. You can use them to specify the types of parameters and/or the return type when you call the function.

```
// The type is correctly inferred because it is defined in the function call
var obj = GetComponent<Rigidbody>();
```

In C#, it can save a lot of keystrokes and cast; for example:

```
Rigidbody rb= (Rigidbody) go.GetComponent(typeof(Rigidbody));
```

Compared to:

```
Rigidbody rb = go.GetComponent<Rigidbody>();
```

Any function that has a generic variant listed on its Scripting API Reference documentation page allows this special calling syntax.

Scripting restrictions

Leave feedback

We strive to provide a common scripting API and experience across all platforms Unity supports. However, some platforms have inherent restrictions. To help you understand these restrictions and support cross-platform code, the following table describes which restrictions apply to each platform and **scripting backend**:

.NET 4.x equivalent scripting runtime

Platform (scripting backend)	*Ahead-of-time compile*	*No threads*	*.NET Core class libraries subset*
Android (IL2CPP)	✔		
Android (Mono)			
iOS (IL2CPP)	✔		
PlayStation 4 (IL2CPP)	✔		
PlayStation Vita (IL2CPP)	✔		
Standalone (IL2CPP)	✔		
Standalone (Mono)			
Switch (IL2CPP)	✔		
Universal Windows Platform (IL2CPP)	✔		
Universal Windows Platform (.NET)			✔
WebGL (IL2CPP)	✔	✔	
WiiU (Mono)			
XBox One (IL2CPP)	✔		

.NET 3.5 equivalent scripting runtime

Warning: This functionality is deprecated, and should no longer be used. Please use .NET 4.

Platform (scripting backend)	*Ahead-of-time compile*	*No threads*	*.NET Core class libraries subset*
Android (IL2CPP)	✔		
Android (Mono)			
iOS (IL2CPP)	✔		
PlayStation 4 (IL2CPP)	✔		
PlayStation 4 (Mono)	✔		
PlayStation Vita (IL2CPP)	✔		
PlayStation Vita (Mono)	✔		
Standalone (IL2CPP)	✔		
Standalone (Mono)			
Switch (IL2CPP)	✔		
Universal Windows Platform (IL2CPP)	✔		
Universal Windows Platform (.NET)			✔
WebGL (IL2CPP)	✔	✔	
WiiU (Mono)			
XBox One (IL2CPP)	✔		

Platform (scripting backend)	*Ahead-of-time compile*	*No threads*	*.NET Core class libraries subset*
XBox One (Mono)	✔		

Ahead-of-time compile

Some platforms do not allow runtime code generation. Therefore, any managed code which depends upon just-in-time (JIT) compilation on the target device will fail. Instead, we need to compile all of the managed code ahead-of-time (AOT). Often, this distinction doesn't matter, but in a few specific cases, AOT platforms require additional consideration.

System.Reflection.Emit

An AOT platform cannot implement any of the methods in the **System.Reflection.Emit** namespace. Note that the rest of **System.Reflection** is acceptable, as long as the compiler can infer that the code used via reflection needs to exist at runtime.

Serialization

AOT platforms may encounter issues with serialization and deserlization due to the use of reflection. If a type or method is only used via reflection as part of serialization or deserialization, the AOT compiler cannot detect that code needs to be generated for the type or method.

Generic virtual methods

Generic methods require the compiler to do some additional work to expand the code written by the developer to the code actually executed on the device. For example, we need different code for **List** with an **int** or a **double**. In the presence of virtual methods, where behavior is determined at runtime rather than compile time, the compiler can easily require runtime code generation in places where it is not entirely obvious from the source code.

Suppose we have the following code, which works exactly as expected on a JIT platform (it prints "Message value: Zero" to the console once):

```
using UnityEngine;
using System;

public class AOTProblemExample : MonoBehaviour, IReceiver
{
    public enum AnyEnum
    {
        Zero,
        One,
    }

    void Start()
    {
        // Subtle trigger: The type of manager *must* be
        // IManager, not Manager, to trigger the AOT problem.
```

```
        IManager manager = new Manager();
        manager.SendMessage(this, AnyEnum.Zero);
    }

    public void OnMessage<T>(T value)
    {
        Debug.LogFormat("Message value: {0}", value);
    }
}

public class Manager : IManager
{
    public void SendMessage<T>(IReceiver target, T value) {
        target.OnMessage(value);
    }
}

public interface IReceiver
{
    void OnMessage<T>(T value);
}

public interface IManager
{
    void SendMessage<T>(IReceiver target, T value);
}
```

When this code is executed on an AOT platform with the **IL2CPP** scripting backend, this exception occurs:

```
ExecutionEngineException: Attempting to call method 'AOTProblemExample::OnMessag
  at Manager.SendMessage[T] (IReceiver target, .T value) [0x00000] in <filename
  at AOTProblemExample.Start () [0x00000] in <filename unknown>:0
```

Likewise, the Mono scripting backend provides this similar exception:

```
  ExecutionEngineException: Attempting to JIT compile method 'Manager:SendMessag
  at AOTProblemExample.Start () [0x00000] in <filename unknown>:0
```

The AOT compiler does not realize that it should generate code for the generic method **OnMessage** with a **T** of **AnyEnum**, so it blissfully continues, skipping this method. When that method is called, and the runtime can't find the proper code to execute, it gives up with this error message.

To work around an AOT issue like this, we can often force the compiler to generate the proper code for us. If we add a method like this to the **AOTProblemExample** class:

```
public void UsedOnlyForAOTCodeGeneration()
{
    // IL2CPP needs only this line.
    OnMessage(AnyEnum.Zero);

    // Mono also needs this line. Note that we are
    // calling directly on the Manager, not the IManager interface.
    new Manager().SendMessage(null, AnyEnum.Zero);

    // Include an exception so we can be sure to know if this method is ever cal
    throw new InvalidOperationException("This method is used for AOT code genera
}
```

When the compiler encounters the explicit call to **OnMessage** with a **T** of **AnyEnum**, it generates the proper code for the runtime to execute. The method **UsedOnlyForAOTCodeGeneration** does not ever need to be called; it just needs to exist for the compiler to see it.

No threads

Some platforms do not support the use of threads, so any managed code that uses the **System.Threading** namespace will fail at runtime. Also, some parts of the .NET class libraries implicitly depend upon threads. An often-used example is the **System.Timers.Timer** class, which depends on support for threads.

2018–08–16 Page amended with no editorial review

Removed Samsung TV support.

.Net 3.5 scripting runtime deprecated in Unity 2018.3

Script Serialization

Leave feedback

Serialization is the automatic process of transforming data structures or object states into a format that Unity can store and reconstruct later. Some of Unity's built-in features use serialization; features such as saving and loading, the **Inspector** window, instantiation, and **Prefabs**. See documentation on Built-in serialization use for background details on all of these.

How you organise data in your Unity project affects how Unity serializes that data and can have a significant impact on the performance of your project. Below is some guidance on serialization in Unity and how to optimize your project for it.

See also documentation on: Serialization Errors, Custom Serlialization, and Built-in Serialization.

Understanding hot reloading

Hot reloading

Hot reloading is the process of creating or editing **scripts** while the Editor is open and applying the script behaviors immediately. You do not have to restart the game or Editor for changes to take effect.

When you change and save a script, Unity hot reloads all the currently loaded script data. It first stores all serializable variables in all loaded scripts and, after loading the scripts, it restores them. All data that is not serializable is lost after a hot reload.

Saving and loading

Unity uses serialization to load and save Scenes, Assets, and AssetBundles to and from your computer's hard drive. This includes data saved in your own scripting API objects such as MonoBehaviour **components** and ScriptableObjects.

Many of the features in the Unity Editor build on top of the core serialization system. Two things to be particularly aware of with serialization are the Inspector window, and hot reloading.

The Inspector window

When you view or change the value of a **GameObject**'s component field in the Inspector window, Unity serializes this data and then displays it in the Inspector window. The Inspector window does not communicate with the Unity Scripting API when it displays the values of a field.

If you use properties in your script, any of the property getters and setters are never called when you view or change values in the Inspector windows as Unity serializes the Inspector window fields directly. This means that: While the values of a field in the Inspector window represent script properties, changes to values in the Inspector window do not call any property getters and setters in your script

Serialization rules

Serializers in Unity run in a real-time game environment. This has a significant impact on performance. As such, serialization in Unity behaves differently to serialization in other programming environments. Outlined below are a number of tips on how to use serialization in Unity.

How to ensure a field in a script is serialized

Ensure it:

Is `public`, or has a SerializeField attribute

Is not `static`

Is not `const`

Is not `readonly`

Has a `fieldtype` that can be serialized
(See Simple field types that can be serialized, below.)

Simple field types that can be serialized

Custom non-abstract, non-generic classes with the `Serializable` attribute
(See How to ensure a custom class can be serialized, below.)

Custom structs with the `Serializable` attribute

References to objects that derive from UnityEngine.Object

Primitive data types (`int`, `float`, `double`, `bool`, `string`, etc.)

Enum types

Certain Unity built-in types: `Vector2`, `Vector3`, `Vector4`, `Rect`, `Quaternion`, `Matrix4x4`, `Color`, `Color32`, `LayerMask`, `AnimationCurve`, `Gradient`, `RectOffset`, `GUIStyle`

Container field types that can be serialized

An array of a simple field type that can be serialized

A `List<T>` of a simple field type that can be serialized

Note: Unity does not support serialization of multilevel types (multidimensional arrays, jagged arrays, and nested container types).
If you want to serialize these, you have two options: wrap the nested type in a class or struct, or use serialization callbacks ISerializationCallbackReceiver to perform custom serialization. For more information, see documentation on Custom Serialization.

How to ensure a custom class can be serialized

Ensure it:

Has the Serializable attribute

Is not abstract

Is not static

Is not generic, though it may inherit from a generic class

To ensure the fields of a custom class or struct are serialized, see How to ensure a field in a script is serialized, above.

When might the serializer behave unexpectedly?

Custom classes behave like structs

With custom classes that are not derived from UnityEngine.Object Unity serializes them inline by value, similar to the way it serializes structs. If you store a reference to an instance of a custom class in several different fields, they become

separate objects when serialized. Then, when Unity deserializes the fields, they contain different distinct objects with identical data.

When you need to serialize a complex object graph with references, do not let Unity automatically serialize the objects. Instead, use `ISerializationCallbackReceiver` to serialize them manually. This prevents Unity from creating multiple objects from object references. For more information, see documentation on ISerializationCallbackReceiver.

This is only true for custom classes. Unity serializes custom classes "inline" because their data becomes part of the complete serialization data for the MonoBehaviour or ScriptableObject they are used in. When fields reference something that is a UnityEngine.Object-derived class, such as `public Camera myCamera`, Unity serializes an actual reference to the **camera** `UnityEngine.Object`. The same occurs in instances of scripts if they are derived from `MonoBehaviour` or `ScriptableObject`, which are both derived from `UnityEngine.Object`.

No support for `null` for custom classes

Consider how many allocations are made when deserializing a `MonoBehaviour` that uses the following script.

```
class Test : MonoBehaviour
{
    public Trouble t;
}
[Serializable]
class Trouble
{
   public Trouble t1;
   public Trouble t2;
   public Trouble t3;
}
```

It wouldn't be strange to expect one allocation: That of the `Test` object. It also wouldn't be strange to expect two allocations: One for the `Test` object and one for a `Trouble` object.

However, Unity actually makes more than a thousand allocations. The serializer does not support null. If it serializes an object, and a field is null, Unity instantiates a new object of that type, and serializes that. Obviously this could lead to infinite cycles, so there is a depth limit of seven levels. At that point Unity stops serializing fields that have types of custom classes, structs, lists, or arrays.

Since so many of Unity's subsystems build on top of the serialization system, this unexpectedly large serialization stream for the `Test MonoBehaviour` causes all these subsystems to perform more slowly than necessary.

No support for polymorphism

If you have a `public Animal[] animals` and you put in an instance of a `Dog`, a `Cat` and a `Giraffe`, after serialization, you have three instances of `Animal`.

One way to deal with this limitation is to realize that it only applies to custom classes, which get serialized inline. References to other `UnityEngine.Objects` get serialized as actual references, and for those, polymorphism does actually work. You would make a `ScriptableObject` derived class or another `MonoBehaviour` derived class, and

reference that. The downside of this is that you need to store that `Monobehaviour` or scriptable object somewhere, and that you cannot serialize it inline efficiently.

The reason for these limitations is that one of the core foundations of the serialization system is that the layout of the datastream for an object is known ahead of time; it depends on the types of the fields of the class, rather than what happens to be stored inside the fields.

Tips

Optimal use of serialization

You can organise your data to ensure you get optimal use from Unity's serialization.

Organise your data with the aim to have Unity serialize the smallest possible set of data. The primary purpose of this is not to save space on your computer's hard drive, but to make sure that you can maintain backwards compatibility with previous versions of the project. Backwards compatibility can become more difficult later on in development if you work with large sets of serialized data.

Organise your data to never have Unity serialize duplicate data or cached data. This causes significant problems for backwards compatibility: It carries a high risk of error because it is too easy for data to get out of sync.

Avoid nested, recursive structures where you reference other classes.The layout of a serialized structure always needs to be the same; independent of the data and only dependent on what is exposed in the script. The only way to reference other objects is through classes derived from UnityEngine.Object. These classes are completely separate; they only reference each other and they don't embed the contents.

Making Editor code hot reloadable

When reloading scripts, Unity serializes and stores all variables in all loaded scripts. After reloading the scripts, Unity then restores them to their original, pre-serialization values.

When reloading scripts, Unity restores all variables - including private variables - that fulfill the requirements for serialization, even if a variable has no SerializeField attribute. In some cases, you specifically need to prevent private variables from being restored: For example, if you want a reference to be null after reloading from scripts. In this case, use the NonSerialized attribute.

Unity never restores static variables, so do not use static variables for states that you need to keep after reloading a script.

• 2017–05–15 Page amended with editorial review

Built-in serialization

Leave feedback

Some of the built-in features of Unity automatically use serialization. These are outlined below.

See the documentation on Script Serialization for further information.

Saving and loading

Unity uses serialization to load and save Scenes, Assets, and AssetBundles to and from your computer's hard drive. This includes data saved in your own scripting API objects such as MonoBehaviour components and ScriptableObjects.

This happens in the Editor's Play Mode and Edit Mode.

Inspector window

When you view or change the value of a GameObject's component field in the Inspector window, Unity serializes this data and then displays it in the **Inspector** window. The **Inspector** window does not communicate with the Unity Scripting API when it displays the values of a field. If you use properties in your script, any of the property getters and setters are never called when you view or change values in the **Inspector** windows as Unity serializes the **Inspector** window fields directly.

Reloading scripts in the Unity Editor

When you change and save a script, Unity reloads all the currently loaded script data. It first stores all serializable variables in all loaded **scripts**, and after loading the **scripts** restores them. All data that is not serializable is lost after the script is reloaded.

This affects all Editor windows, as well as all MonoBehaviours in the project. Unlike other cases of serialization in Unity, private fields are serialized by default when reloading, even if they don't have the 'SerializeField' attribute.

Prefabs

In the context of serialization, a Prefab is the serialized data of one or more GameObjects and components. A Prefab instance contains a reference to both the Prefab source and a list of modifications to it. The modifications are what Unity needs to do to the Prefab source to create that particular Prefab instance.

The Prefab instance only exists while you edit your project in the Unity Editor. During the project build, the Unity Editor instantiates a GameObject from its two sets of serialization data: the Prefab source and the Prefab instance's modifications.

Instantiation

When you call Instantiate on anything that exists in a Scene, such as a Prefab or a GameObjects, Unity serializes it. This happens both at runtime and in the Editor. Everything that derives from UnityEngine.Object can be serialized.

Unity then creates a new GameObject and deserializes the data onto the new GameObject. Next, Unity runs the same serialization code in a different variant to report which other `UnityEngine.Objects` are being referenced. It checks all referenced `UnityEngine.Objects` to see if they are part of the data being instantiated. If the reference points to something "external", such as a Texture, Unity keeps that reference as it is. If the reference points to something "internal", such as a child GameObject, Unity patches the reference to the corresponding copy.

Unloading unused assets

`Resource.GarbageCollectSharedAssets()` is the native Unity garbage collector and performs a different function to the standard C# garbage collector. It runs after you load a Scene and checks for objects (like textures) that are no longer referenced and unloads them safely. The native Unity garbage collector runs the serializer in a variation in which objects report all references to external `UnityEngine.Objects`. This is how Textures that were used by one scene are unloaded in the next.

• 2017–05–15 Page published with editorial review

Custom serialization

Leave feedback

Serialization is the automatic process of transforming data structures or object states into a format that Unity can store and reconstruct later. (See the documentation on Script Serialization for further information on Unity's serialization.)

Sometimes you might want to serialize something that Unity's serializer doesn't support. In many cases the best approach is to use serialization callbacks. (See Unity's Scripting API Reference: ISerializationCallbackReceiver for further information on custom serialization using serlialization callbacks.)

Serialization callbacks allow you to be notified before the serializer reads data from your fields and after it has finished writing to them. You can use serialization callbacks to give your hard-to-serialize data a different representation at runtime to its representation when you actually serialize.

To do this, transform your data into something Unity understands right before Unity wants to serialize it. Then, right after Unity has written the data to your fields, you can transform the serialized form back into the form you want to have your data in at runtime

For example: You want to have a tree data structure. If you let Unity directly serialize the data structure, the "no support for null" limitation would cause your data stream to become very big, leading to performance degradations in many systems. This is shown in Example 1, below.

Example 1: Unity's direct serlialization, leading to performance issues

```
using UnityEngine;
using System.Collections.Generic;
using System;

public class VerySlowBehaviourDoNotDoThis : MonoBehaviour {
    [Serializable]
    public class Node {
        public string interestingValue = "value";
        //The field below is what makes the serialization data become huge becau
        //it introduces a 'class cycle'.
        public List<Node> children = new List<Node>();
    }
    //this gets serialized
    public Node root = new Node();
    void OnGUI() {
        Display (root);
    }
    void Display(Node node) {
        GUILayout.Label ("Value: ");
        node.interestingValue = GUILayout.TextField(node.interestingValue, GUILa
        GUILayout.BeginHorizontal ();
        GUILayout.Space (20);
```

```
        GUILayout.BeginVertical ();
        foreach (var child in node.children)
        Display (child);
        if (GUILayout.Button ("Add child"))
        node.children.Add (new Node ());
        GUILayout.EndVertical ();
        GUILayout.EndHorizontal ();
    }
}
```

Instead, you tell Unity not to serialize the tree directly, and you make a separate field to store the tree in a serialized format, suited to Unity's serializer. This is shown in Example 2, below.

Example 2: Avoiding Unity's direct serlialization and avoiding performance issues

```
using System.Collections.Generic;
using System;

public class BehaviourWithTree : MonoBehaviour, ISerializationCallbackReceiver {
    // Node class that is used at runtime.
    // This is internal to the BehaviourWithTree class and is not serialized.
    public class Node {
        public string interestingValue = "value";
        public List<Node> children = new List<Node>();
    }
    // Node class that we will use for serialization.
    [Serializable]
    public struct SerializableNode {
        public string interestingValue;
        public int childCount;
        public int indexOfFirstChild;
    }
    // The root node used for runtime tree representation. Not serialized.
    Node root = new Node();
    // This is the field we give Unity to serialize.
    public List<SerializableNode> serializedNodes;
    public void OnBeforeSerialize() {
        // Unity is about to read the serializedNodes field's contents.
        // The correct data must now be written into that field "just in time".
        if (serializedNodes == null) serializedNodes = new List<SerializableNode
        if (root == null) root = new Node ();
        serializedNodes.Clear();
```

```
        AddNodeToSerializedNodes(root);
        // Now Unity is free to serialize this field, and we should get back the
        // data when it is deserialized later.
    }
    void AddNodeToSerializedNodes(Node n) {
        var serializedNode = new SerializableNode () {
            interestingValue = n.interestingValue,
            childCount = n.children.Count,
            indexOfFirstChild = serializedNodes.Count+1
        }
        ;
        serializedNodes.Add (serializedNode);
        foreach (var child in n.children)
        AddNodeToSerializedNodes (child);
    }
    public void OnAfterDeserialize() {
        //Unity has just written new data into the serializedNodes field.
        //let's populate our actual runtime data with those new values.
        if (serializedNodes.Count > 0) {
            ReadNodeFromSerializedNodes (0, out root);
        } else
        root = new Node ();
    }
    int ReadNodeFromSerializedNodes(int index, out Node node) {
        var serializedNode = serializedNodes [index];
        // Transfer the deserialized data into the internal Node class
        Node newNode = new Node() {
            interestingValue = serializedNode.interestingValue,
            children = new List<Node> ()
        }
        ;
        // The tree needs to be read in depth-first, since that's how we wrote i
        for (int i = 0; i != serializedNode.childCount; i++) {
            Node childNode;
            index = ReadNodeFromSerializedNodes (++index, out childNode);
            newNode.children.Add (childNode);
        }
        node = newNode;
        return index;
    }
    // This OnGUI draws out the node tree in the Game View, with buttons to add
    void OnGUI() {
        if (root != null)
        Display (root);
    }
    void Display(Node node) {
```

```
        GUILayout.Label ("Value: ");
        // Allow modification of the node's "interesting value".
        node.interestingValue = GUILayout.TextField(node.interestingValue, GUILa
        GUILayout.BeginHorizontal ();
        GUILayout.Space (20);
        GUILayout.BeginVertical ();
        foreach (var child in node.children)
        Display (child);
        if (GUILayout.Button ("Add child"))
        node.children.Add (new Node ());
        GUILayout.EndVertical ();
        GUILayout.EndHorizontal ();
    }
}
```

• 2017–05–15 Page published with editorial review

Script serialization errors

Leave feedback

Serialization is the automatic process of transforming data structures or object states into a format that Unity can store and reconstruct later. (See the documentation on Script Serialization for further information.)

In certain circumstances, Script serialization can cause errors. Fixes to some of these are listed below.

Calling Unity Scripting API from constructor or field initializers

Calling Scripting API such as GameObject.Find inside a MonoBehaviour constructor or field initializer triggers the error: “Find is not allowed to be called from a MonoBehaviour constructor (or instance field initializer), call in in Awake or Start instead.”

Fix this by making the call to the Scripting API in MonoBehaviour.Start instead of in the constructor.

Calling Unity Scripting API during deserialization

Calling Scripting API such as GameObject.Find from within the constructor of a class marked with `System.Serializable` triggers the error: “Find is not allowed to be called during serialization, call it from Awake or Start instead.”

To fix this, edit your code so that it makes no Scripting API calls in any constructors for any serialized objects.

Thread-safe Unity Scripting API

The majority of the Scripting API is affected by the restrictions listed above. Only select parts of the Unity scripting API are exempt and may be called from anywhere. These are:

Debug.Log

Mathf functions

Simple self-contained structs; for example math structs like Vector3 and Quaternion

To reduce the risk of errors during serialization, only call API methods that are self-contained and do not need to get or set data in Unity itself. Only call these if there is no alternative.

• 2017–05–15 Page published with editorial review

UnityEvents

Leave feedback

UnityEvents are a way of allowing user driven callback to be persisted from edit time to run time without the need for additional programming and script configuration.

UnityEvents are useful for a number of things:

Content driven callbacks

Decoupling systems

Persistent callbacks

Preconfigured call events

`UnityEvent`s can be added to any `MonoBehaviour` and are executed from code like a standard .net delegate. When a `UnityEvent` is added to a `MonoBehaviour` it appears in the Inspector and persistent callbacks can be added.

`UnityEvent`s have similar limitations to standard delegates. That is, they hold references to the element that is the target and this stops the target being garbage collected. If you have a UnityEngine.Object as the target and the native representation disappears the callback will not be invoked.

Using UnityEvents

To configure a callback in the editor there are a few steps to take:

Make sure your script imports/uses `UnityEngine.Events`.

Select the + icon to add a slot for a callback

Select the UnityEngine.Object you wish to receive the callback (You can use the object selector for this)

Select the function you wish to be called

You can add more then one callback for the event

When configuring a `UnityEvent` in the **Inspector** there are two types of function calls that are supported:

Static. Static calls are preconfigured calls, with preconfigured values that are set in the **UI**. This means that when the callback is invoked, the target function is invoked with the argument that has been entered into the **UI**.
Dynamic. Dynamic calls are invoked using an argument that is sent from code, and this is bound to the type of UnityEvent that is being invoked. The UI filters the callbacks and only shows the dynamic calls that are valid for the UnityEvent.

Generic UnityEvents

By default a `UnityEvent` in a `Monobehaviour` binds dynamically to a void function. This does not have to be the case as dynamic invocation of UnityEvents supports binding to functions with up to 4 arguments. To do this you need to define a custom `UnityEvent` class that supports multiple arguments. This is quite easy to do:

```
[Serializable]

public class StringEvent : UnityEvent <string> {}
```

By adding an instance of this to your class instead of the base `UnityEvent` it will allow the callback to bind dynamically to string functions.

This can then be invoked by calling the `Invoke()` function with a `string` as argument.

UnityEvents can be defined with up to 4 arguments in their generic definition.

What is a Null Reference Exception?

Leave feedback

A `NullReferenceException` happens when you try to access a reference variable that isn't referencing any object. If a reference variable isn't referencing an object, then it'll be treated as `null`. The run-time will tell you that you are trying to access an object, when the variable is `null` by issuing a `NullReferenceException`.

Reference variables in c# and JavaScript are similar in concept to pointers in C and C++. Reference types default to `null` to indicate that they are not referencing any object. Hence, if you try and access the object that is being referenced and there isn't one, you will get a `NullReferenceException`.

When you get a `NullReferenceException` in your code it means that you have forgotten to set a variable before using it. The error message will look something like:

```
NullReferenceException: Object reference not set to an instance of an object
  at Example.Start () [0x0000b] in /Unity/projects/nre/__Assets__/Example.cs:10
```

This error message says that a `NullReferenceException` happened on line 10 of the script file `Example.cs`. Also, the message says that the exception happened inside the `Start()` function. This makes the Null Reference Exception easy to find and fix. In this example, the code is:

```
//c# example
using UnityEngine;
using System.Collections;

public class Example : MonoBehaviour {

    // Use this for initialization
    void Start () {
        __GameObject__ go = __GameObject__.Find("wibble");
        Debug.Log(go.name);
    }

}
```

The code simply looks for a game object called "wibble". In this example there is no game object with that name, so the `Find()` function returns `null`. On the next line (line 9) we use the `go` variable and try and print out the name of the game object it references. Because we are accessing a game object that doesn't exist the run-time gives us a `NullReferenceException`

Null Checks

Although it can be frustrating when this happens it just means the script needs to be more careful. The solution in this simple example is to change the code like this:

```
using UnityEngine;
using System.Collections;
```

```
public class Example : MonoBehaviour {

    void Start () {
        GameObject go = GameObject.Find("wibble");
        if (go) {
            Debug.Log(go.name);
        } else {
            Debug.Log("No game object called wibble found");
        }
    }

}
```

Now, before we try and do anything with the `go` variable, we check to see that it is not `null`. If it it `null`, then we display a message.

Try/Catch Blocks

Another cause for `NullReferenceException` is to use a variable that should be initialised in the **Inspector**. If you forget to do this, then the variable will be `null`. A different way to deal with `NullReferenceException` is to use try/catch block. For example, this code:

```
using UnityEngine;
using System;
using System.Collections;

public class Example2 : MonoBehaviour {

    public Light myLight; // set in the inspector

    void Start () {
        try {
            myLight.color = Color.yellow;
        }
        catch (NullReferenceException ex) {
            Debug.Log("myLight was not set in the inspector");
        }
    }

}
```

In this code example, the variable called `myLight` is a `Light` which should be set in the Inspector window. If this variable is not set, then it will default to `null`. Attempting to change the color of the light in the `try` block causes a `NullReferenceException` which is picked up by the `catch` block. The `catch` block displays a message which might be more helpful to artists and game designers, and reminds them to set the light in the inspector.

Summary

`NullReferenceException` happens when your script code tries to use a variable which isn't set (referencing) and object.
The error message that appears tells you a great deal about where in the code the problem happens.
`NullReferenceException` can be avoided by writing code that checks for `null` before accessing an object, or uses try/catch blocks.

Important Classes

Leave feedback

These are some of the most important classes you'll be using when scripting with Unity. They cover some of the core areas of Unity's scriptable systems and provide a good starting point for looking up which functions and events are available.

Class:	*Description:*
MonoBehaviour	The base class for all new Unity **scripts**, the MonoBehaviour reference provides you with a list of all the functions and events that are available to standard **scripts** attached to Game Objects. Start here if you're looking for any kind of interaction or control over individual objects in your game.
Transform	Every Game Object has a position, rotation and scale in space (whether 3D or 2D), and this is represented by the Transform **component**. As well as providing this information, the transform **component** has many helpful functions which can be used to move, scale, rotate, reparent and manipulate objects, as well as converting coordinates from one space to another.
Rigidbody / Rigidbody2D	For most gameplay elements, the **physics engine** provides the easiest set of tools for moving objects around, detecting triggers and **collisions**, and applying forces. The Rigidbody class (or its 2D equivalent, Rigidbody2D) provides all the properties and functions you'll need to play with velocity, mass, drag, force, torque, collision and more.

Vector Cookbook

Leave feedback

Although vector operations are easy to describe, they are surprisingly subtle and powerful and have many uses in games programming. The following pages offer some suggestions about using vectors effectively in your code.

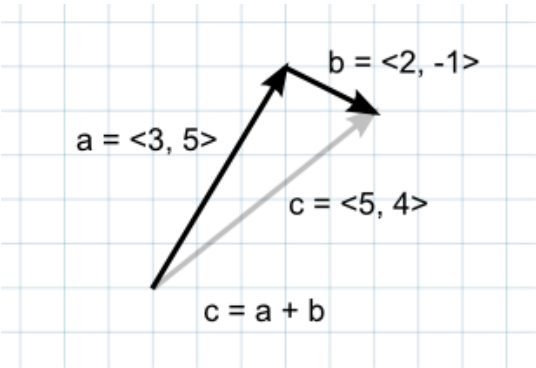
Understanding Vector Arithmetic

Leave feedback

Vector arithmetic is fundamental to 3D graphics, physics and animation and it is useful to understand it in depth to get the most out of Unity. Below are descriptions of the main operations and some suggestions about the many things they can be used for.

Addition

When two vectors are added together, the result is equivalent to taking the original vectors as “steps”, one after the other. Note that the order of the two parameters doesn’t matter, since the result is the same either way.



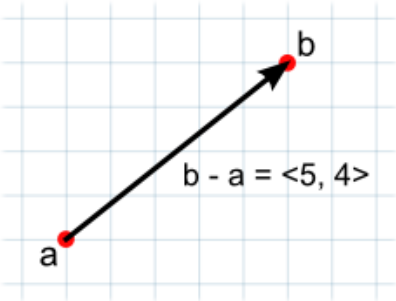
If the first vector is taken as a point in space then the second can be interpreted as an offset or “jump” from that position. For example, to find a point 5 units above a location on the ground, you could use the following calculation:-

```
var pointInAir = pointOnGround + new Vector3(0, 5, 0);
```

If the vectors represent forces then it is more intuitive to think of them in terms of their direction and magnitude (the magnitude indicates the size of the force). Adding two force vectors results in a new vector equivalent to the combination of the forces. This concept is often useful when applying forces with several separate **components** acting at once (eg, a rocket being propelled forward may also be affected by a crosswind).

Subtraction

Vector subtraction is most often used to get the direction and distance from one object to another. Note that the order of the two parameters **does** matter with subtraction:-




```
// The vector d has the same magnitude as c but points in the opposite direction.
var c = b - a;
var d = a - b;
```

As with numbers, adding the negative of a vector is the same as subtracting the positive.

```
// These both give the same result.
var c = a - b;
var c = a + -b;
```

The negative of a vector has the same magnitude as the original and points along the same line but in the exact opposite direction.

Scalar Multiplication and Division

When discussing vectors, it is common to refer to an ordinary number (eg, a float value) as a scalar. The meaning of this is that a scalar only has “scale” or magnitude whereas a vector has both magnitude and direction.

Multiplying a vector by a scalar results in a vector that points in the same direction as the original. However, the new vector’s magnitude is equal to the original magnitude multiplied by the scalar value.

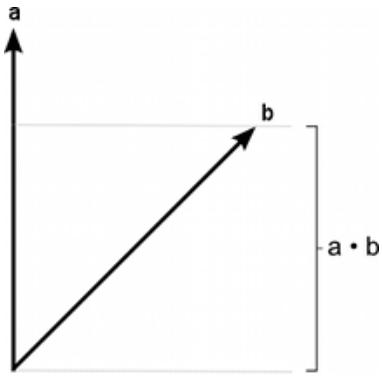
Likewise, scalar division divides the original vector’s magnitude by the scalar.

These operations are useful when the vector represents a movement offset or a force. They allow you to change the magnitude of the vector without affecting its direction.

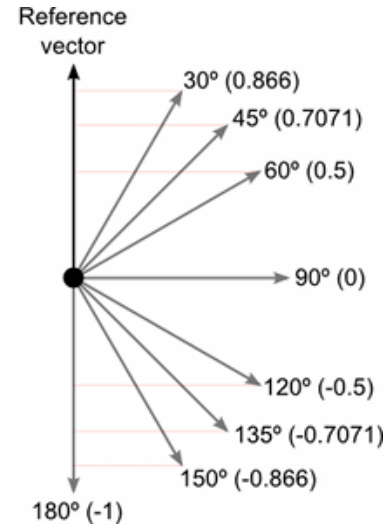
When any vector is divided by its own magnitude, the result is a vector with a magnitude of 1, which is known as a normalized vector. If a normalized vector is multiplied by a scalar then the magnitude of the result will be equal to that scalar value. This is useful when the direction of a force is constant but the strength is controllable (eg, the force from a car’s wheel always pushes forwards but the power is controlled by the driver).

Dot Product

The dot product takes two vectors and returns a scalar. This scalar is equal to the magnitudes of the two vectors multiplied together and the result multiplied by the cosine of the angle between the vectors. When both vectors are normalized, the cosine essentially states how far the first vector extends in the second’s direction (or vice-versa - the order of the parameters doesn’t matter).



It is easy enough to think in terms of angles and then find the corresponding cosines using a calculator. However, it is useful to get an intuitive understanding of some of the main cosine values as shown in the diagram below:-

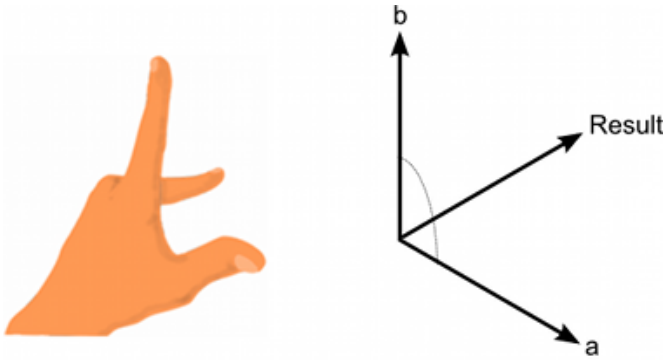


The dot product is a very simple operation that can be used in place of the Mathf.Cos function or the vector magnitude operation in some circumstances (it doesn't do exactly the same thing but sometimes the effect is equivalent). However, calculating the dot product function takes much less CPU time and so it can be a valuable optimization.

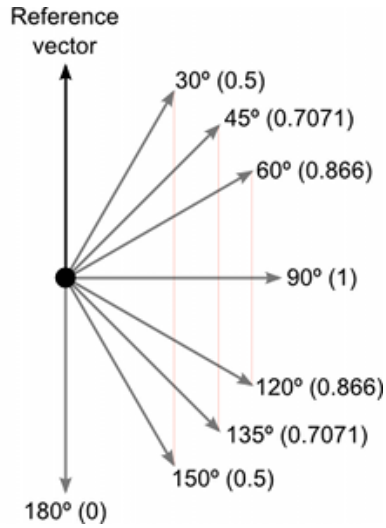
Cross Product

The other operations are defined for 2D and 3D vectors and indeed vectors with any number of dimensions. The cross product, by contrast, is only meaningful for 3D vectors. It takes two vectors as input and returns another vector as its result.

The result vector is perpendicular to the two input vectors. The "left hand rule" can be used to remember the direction of the output vector from the ordering of the input vectors. If the first parameter is matched up to the thumb of the hand and the second parameter to the forefinger, then the result will point in the direction of the middle finger. If the order of the parameters is reversed then the resulting vector will point in the exact opposite direction but will have the same magnitude.



The magnitude of the result is equal to the magnitudes of the input vectors multiplied together and then that value multiplied by the sine of the angle between them. Some useful values of the sine function are shown below:-



The cross product can seem complicated since it combines several useful pieces of information in its return value. However, like the dot product, it is very efficient mathematically and can be used to optimize code that would otherwise depend on slow transcendental functions.

Direction and Distance from One Object to Another

Leave feedback

If one point in space is subtracted from another, then the result is a vector that “points” from one object to the other:

```
// Gets a vector that points from the player's position to the target's.
var heading = target.position - player.position;
```

As well as pointing in the direction of the target object, this vector’s magnitude is equal to the distance between the two positions. It is common to need a normalized vector giving the direction to the target and also the distance to the target (say for directing a projectile). The distance between the objects is equal to the magnitude of the heading vector and this vector can be normalized by dividing it by its magnitude:-

```
var distance = heading.magnitude;
var direction = heading / distance; // This is now the normalized direction.
```

This approach is preferable to using both the magnitude and normalized properties separately, since they are both quite CPU-hungry (they both involve calculating a square root).

If you only need to use the distance for comparison (for a proximity check, say) then you can avoid the magnitude calculation altogether. The sqrMagnitude property gives the square of the magnitude value, and is calculated like the magnitude but without the time-consuming square root operation. Rather than compare the magnitude against a known distance, you can compare the squared magnitude against the squared distance:-

```
if (heading.sqrMagnitude < maxRange * maxRange) {
    // Target is within range.
}
```

This is much more efficient than using the true magnitude in the comparison.

Sometimes, the overground heading to a target is required. For example, imagine a player standing on the ground who needs to approach a target floating in the air. If you subtract the player’s position from the target’s then the resulting vector will point upwards towards the target. This is not suitable for orienting the player’s

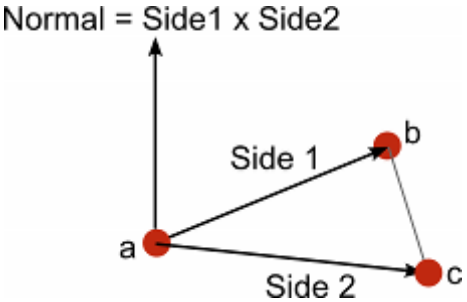
transform since they will also point upwards; what is really needed is a vector from the player's position to the position on the ground directly below the target. This is easily obtained by taking the result of the subtraction and setting the Y coordinate to zero:-

```
var heading = target.position - player.position;
heading.y = 0;  // This is the overground heading.
```

Computing a Normal/Perpendicular vector

Leave feedback

A normal vector (ie, a vector perpendicular to a plane) is required frequently during **mesh** generation and may also be useful in path following and other situations. Given three points in the plane, say the corner points of a **mesh** triangle, it is easy to find the normal. Pick any of the three points and then subtract it from each of the two other points separately to give two vectors:-




```
var a: Vector3;
var b: Vector3;
var c: Vector3;

var side1: Vector3 = b - a;
var side2: Vector3 = c - a;
```

The cross product of these two vectors will give a third vector which is perpendicular to the surface. The “left hand rule” can be used to decide the order in which the two vectors should be passed to the cross product function. As you look down at the top side of the surface (from which the normal will point outwards) the first vector should sweep around clockwise to the second:-

```
var perp: Vector3 = Vector3.Cross(side1, side2);
```

The result will point in exactly the opposite direction if the order of the input vectors is reversed.

For meshes, the normal vector must also be normalized. This can be done with the normalized property, but there is another trick which is occasionally useful. You can also normalize the perpendicular vector by dividing it by its magnitude:-

```
var perpLength = perp.magnitude;
perp /= perpLength;
```

It turns out that the area of the triangle is equal to perpLength / 2. This is useful if you need to find the surface area of the whole mesh or want to choose triangles randomly with probability based on their relative areas.

The Amount of One Vector's Magnitude that Lies in Another Vector's Direction

Leave feedback

A car's speedometer typically works by measuring the rotational speed of the wheels. The car may not be moving directly forward (it may be skidding sideways, for example) in which case part of the motion will not be in the direction the speedometer can measure. The magnitude of an object's **rigidbody**.velocity vector will give the speed in its direction of overall motion but to isolate the speed in the forward direction, you should use the dot product:-

```
var fwdSpeed = Vector3.Dot(rigidbody.velocity, transform.forward);
```

Naturally, the direction can be anything you like but the direction vector must always be normalized for this calculation. Not only is the result more correct than the magnitude of the velocity, it also avoids the slow square root operation involved in finding the magnitude.

Scripting Tools

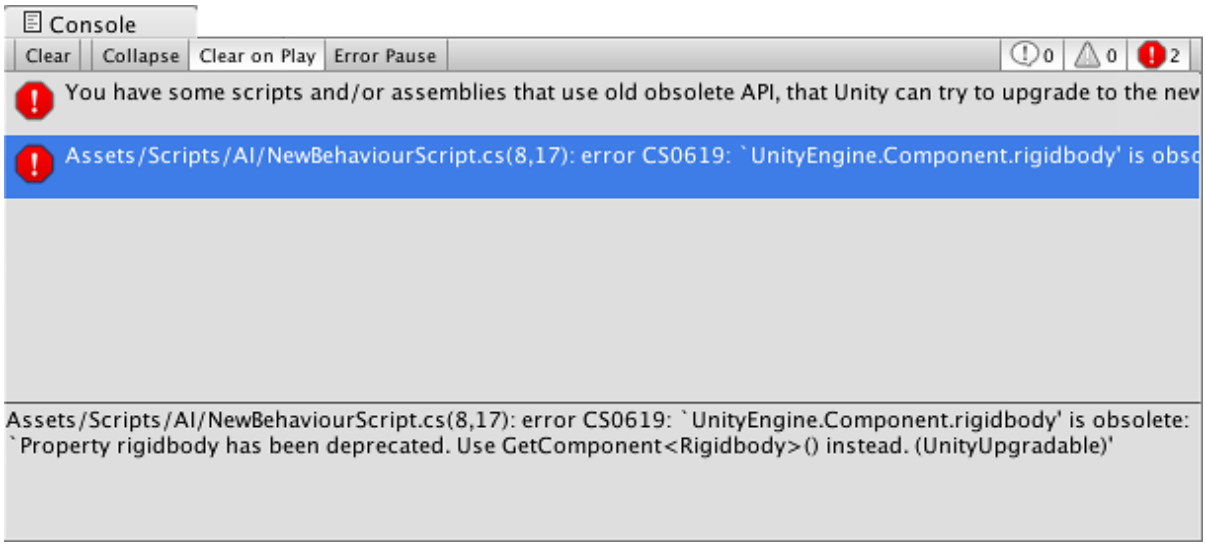
Leave feedback

This section covers the tools within the Unity Editor, and tools supplied with Unity that assist you in developing your **scripts**.

Console Window

Leave feedback

The **Console** Window (menu: **Window** > **General** > **Console**) shows errors, warnings and other messages generated by Unity. To aid with debugging, you can also show your own messages in the Console using the Debug.Log, Debug.LogWarning and Debug.LogError functions.



The **toolbar** of the **console window** has a number of options that affect how messages are displayed.

The *Clear* button removes any messages generated from your code but retains compiler errors. You can arrange for the console to be cleared automatically whenever you run the game by enabling the *Clear On Play* option.

You can also change the way messages are shown and updated in the console. The *Collapse* option shows only the first instance of an error message that keeps recurring. This is very useful for runtime errors, such as null references, that are sometimes generated identically on each frame update. The *Error Pause* option will cause playback to be paused whenever Debug.LogError is called from a script (but note that Debug.Log will not pause in this way). This can be handy when you want to freeze playback at a specific point in execution and inspect the **scene**.

Finally, there are two options for viewing additional information about errors. The *Open Player Log* and *Open Editor Log* items on the console tab menu access Unity's log files which record details that may not be shown in the console. See the page about Log Files for further information.

Obsolete API Warnings and Automatic Updates

Among other messages, Unity shows warnings about the usage of obsolete API calls in your code. For example, Unity once had "shortcuts" in MonoBehaviour and other classes to access common **component** types. So, for example, you could access a Rigidbody on the object using code like:

```
// The "rigidbody" variable is part of the class and not declared in the user
Vector3 v = rigidbody.velocity;
```

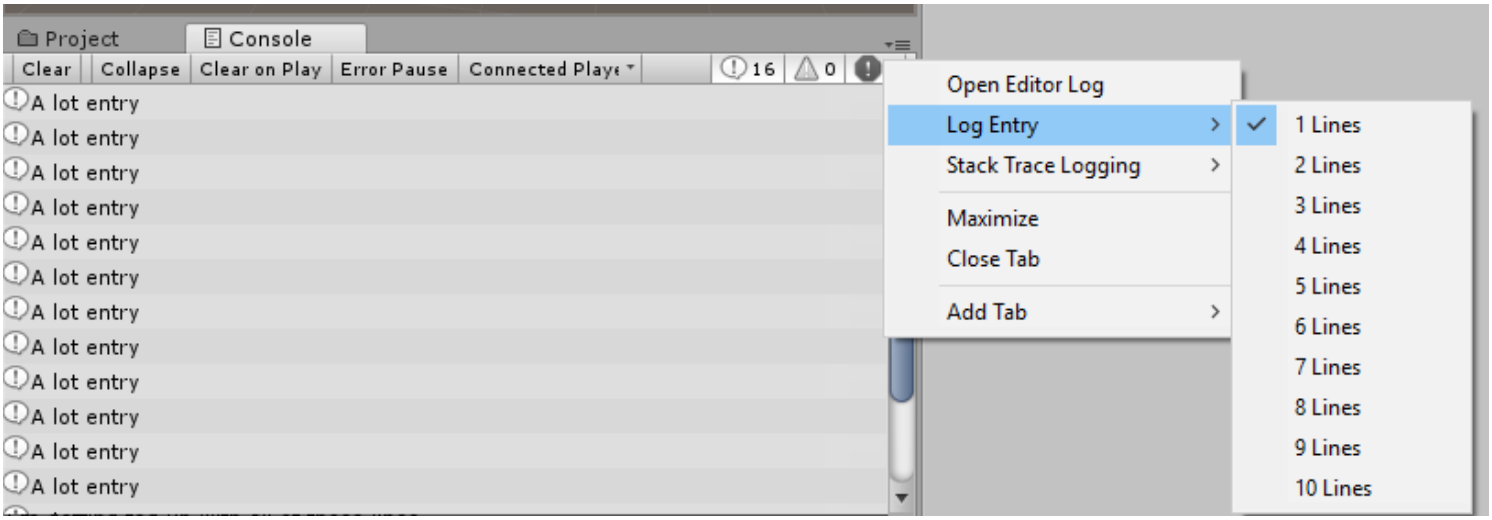
These shortcuts have been deprecated, so you should now use code like:

```
// Use GetComponent to access the component.
Rigidbody rb = GetComponent<Rigidbody>();
Vector3 v = rb.velocity;
```

When obsolete API calls are detected, Unity will show a warning message about them. When you double-click this message, Unity will attempt to upgrade the deprecated usage to the recommended equivalent automatically.

Adjusting the line count

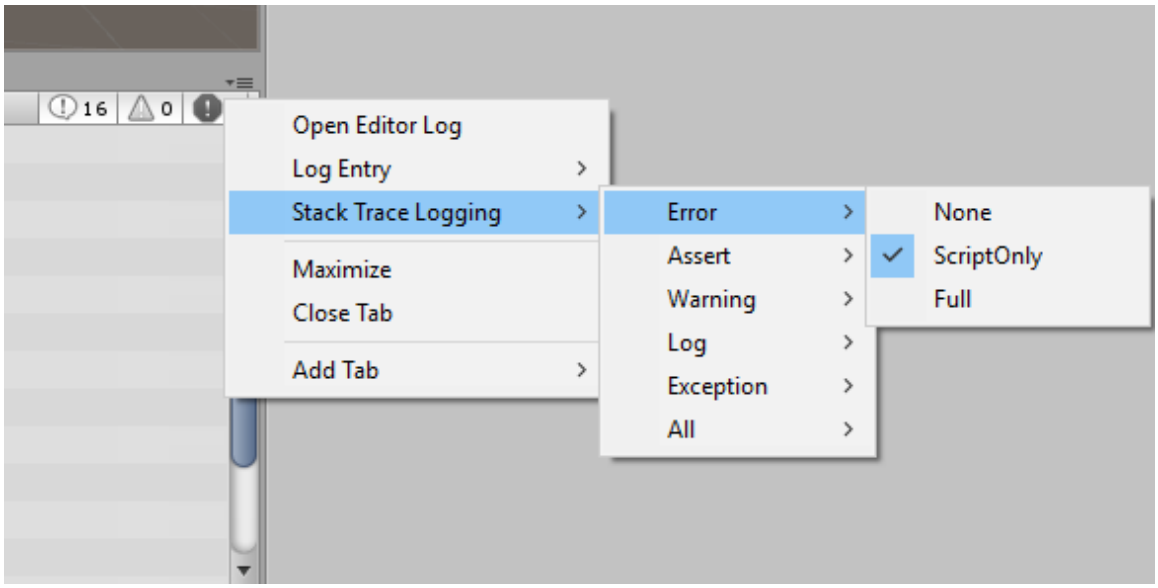
To adjust the number of lines that a log entry displays in the list, click the exclamation button, go to **Log Entry**, and choose the number of lines.



This allows you to set the granularity you want for the window in terms of the amount of context versus how many entries fit in the window.

Stack trace logging

You can specify how accurate stack trace should be captured when log message is printed to the console or log file.



Log entry line count

This is especially helpful when the error message is not very clear, by looking at the stack trace, you can deduct from what engine area the error appears. There are three options for logging stack trace:

None - stack trace won't be printed
ScriptOnly - only managed stack trace will be printed
Full - both native and managed stack trace will be printed, note - resolving full stack trace is an expensive operation and should be only used for debugging purposes

You can also control stack trace logging via scripting API. See API reference documentation on Application.stackTraceLogType for more details.

2017–09–18 Page amended with limited editorial review

Log entry line count added in 2017.3

Log Files

Leave feedback

There might be times during development when you need to obtain information from the logs of the standalone player you've built, the target device, or the Editor. Usually you need to see these files when you have experienced a problem, to find out exactly where the problem occurred.

On macOS, the player and Editor logs can be accessed uniformly through the standard **Console.app** utility.

On Windows, the Editor logs are placed in folders which are not shown in the Windows Explorer by default. See below.

Editor

To view the Editor log, select **Open Editor Log** in Unity's **Console window**.

OS	Log files
macOS	`~/Library/Logs/Unity/Editor.log`
Windows	`C:\Users\username\AppData\Local\Unity\Editor\Editor.log`

On macOS, all the logs can be accessed uniformly through the standard **Console.app** utility.

On Windows, the Editor log file is stored in the local application data folder <LOCALAPPDATA>\Unity\Editor\Editor.log, where <LOCALAPPDATA> is defined by CSIDL_LOCAL_APPDATA.

Player

OS	Log files
macOS	`~/Library/Logs/Unity/Player.log`
Windows	`C:\Users\username\AppData\LocalLow\CompanyName\ProductName\output_log.txt`
Linux	`~/.config/unity3d/CompanyName/ProductName/Player.log`

Note that on Windows and Linux standalones, the location of the log file can be changed (or logging suppressed). See documenttion on Command line arguments for further details.

iOS

Access the device log in XCode via the GDB console or the Organizer Console. The latter is useful for getting crashlogs when your application was not running through the XCode debugger.

The Troubleshooting and Reporting crash bugs guides may be useful for you.

Android

Access the device log using the logcat console. Use the **adb** application found in **Android SDK/platform-tools directory** with a trailing **logcat** parameter:

```
$ adb logcat
```

Another way to inspect the LogCat is to use the Dalvik Debug Monitor Server (DDMS). DDMS can be started either from **Eclipse** or from inside the **Android SDK/tools**. DDMS also provides a number of other debug related tools.

Universal Windows Platform

Device	Log files
Desktop	`%USERPROFILE%\AppData\Local\Packages<productname>\TempState\UnityPlayer.log`

Device	Log files
Windows Phone	Can be retrieved with Windows Phone Power Tools. The Windows Phone IsoStoreSpy can also be helpful.
WebGL	

On WebGL, log output is written to the browser's JavaScript console.

Accessing log files on Windows

On Windows, the log files are stored in locations that are hidden by default.

On Windows Vista/7, make the AppData folder visible in Windows Explorer using **Tools** > **Folder Options...** > **View (tab)**. The Tools menu is hidden by default; press the Alt key once to display it.

• 2017–05–16 Page amended with no editorial review

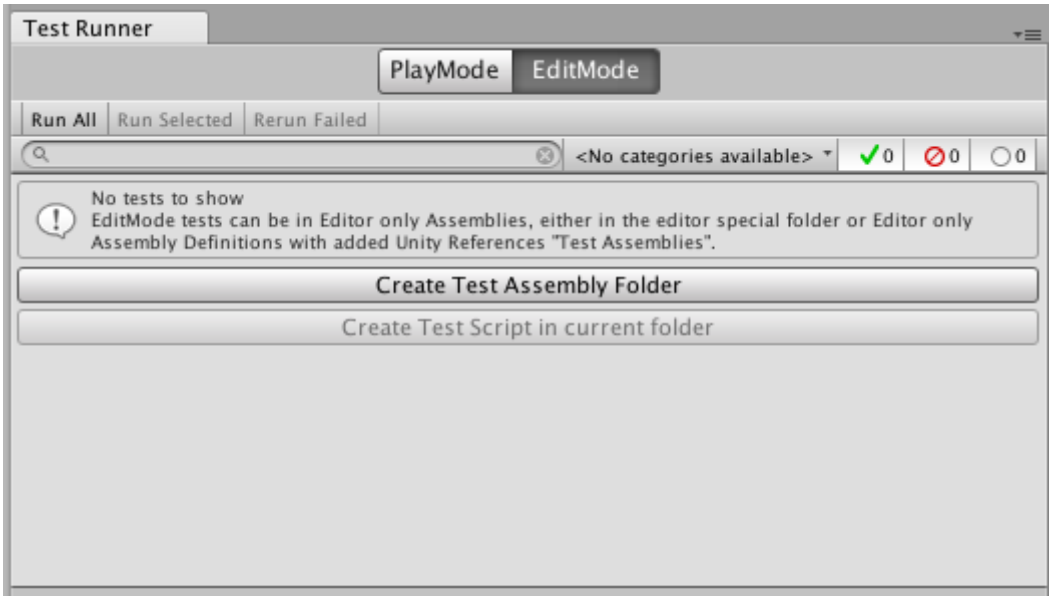
Tizen support discontinued in 2017.3

Unity Test Runner

Leave feedback

The Unity Test Runner is a tool that tests your code in both **Edit** mode and **Play** mode, and also on target platforms such as Standalone, Android, or **iOS**.

To access the Unity Test Runner, go to **Window** > **General** > **Test Runner**.



The Unity Test Runner uses a Unity integration of the NUnit library, which is an open-source unit testing library for .Net languages. For more information about NUnit, see the official NUnit website and the NUnit documentation on GitHub.

UnityTestAttribute is the main addition to the standard NUnit library for the Unity Test Runner. This is a type of unit test that allows you to skip a frame from within a test (which allows background tasks to finish). To use `UnityTestAttribute`:

In **Play** mode: Execute `UnityTestAttribute` as a coroutine.

In **Edit** mode: Execute `UnityTestAttribute` in the EditorApplication.update callback loop.

Known issues and limitations

There are some known issues and limitations of the Unity Test Runner:

The **WebGL** and WSA platforms do not support `UnityTestAttribute`.

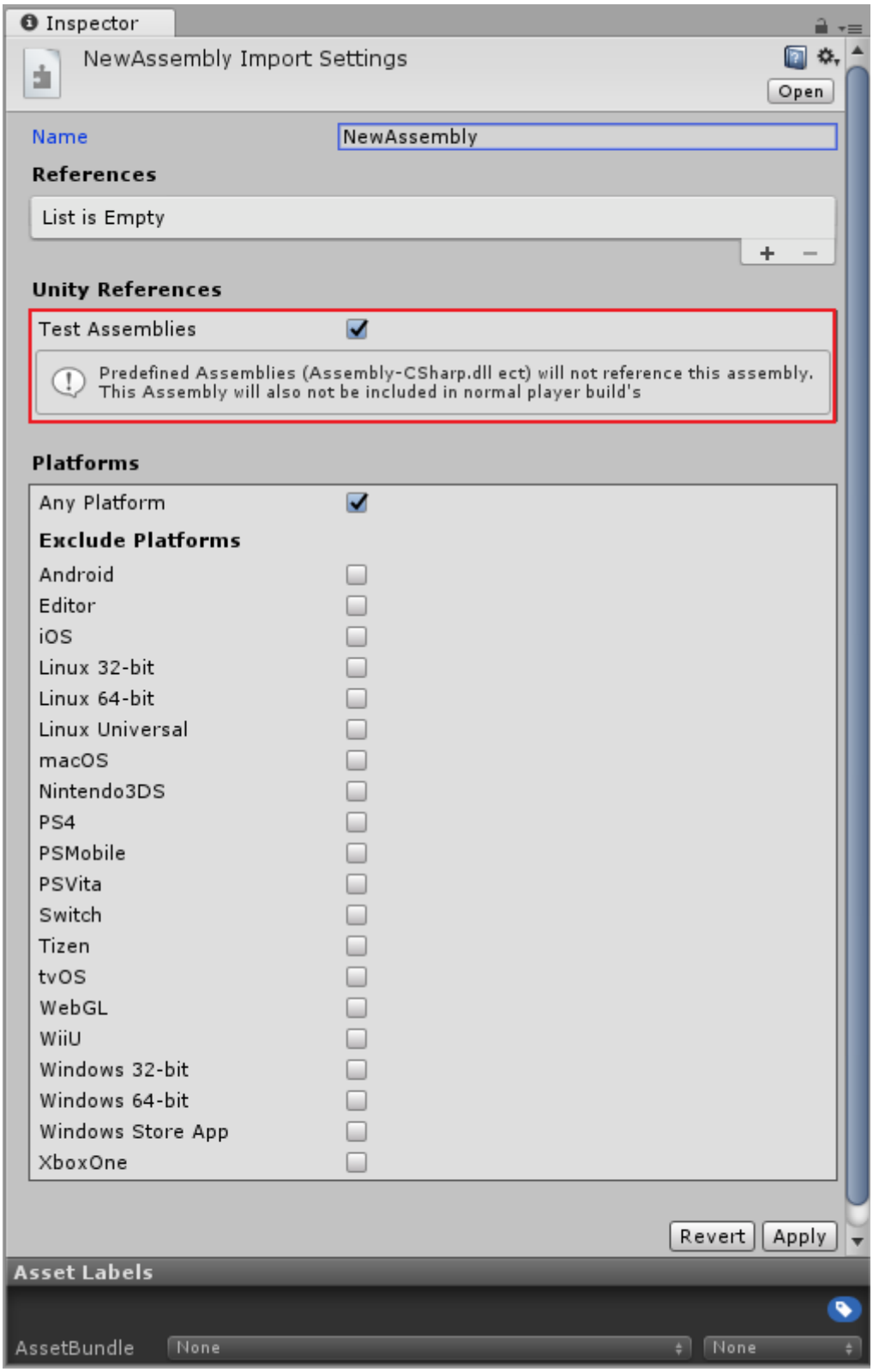
`UnityTest` does not support Parameterized tests (except for `ValueSource`).

How to use Unity Test Runner

This page assumes you already have an understanding of unit testing and NUnit. If you are new to NUnit or would like more information, see to the NUnit documentation on GitHub.

To open the Unity Test Runner, open Unity and go to **Window** > **General** > **Test Runner**. If there are no tests in your Project, click the **Create Test Script in current folder** button to create a basic test script. This button is greyed out if adding a test script would result in a compilation error. The conditions for adding a test script are in

the Editor folder, or any folders using Assembly Definition files that reference test assemblies (NUnit, Unity Test Runner, and user script assemblies).



You can also create test scripts by navigating to **Assets** > **Create** > **C# Test Script**. This option is disabled if adding a test script would result in a compilation error.

Note: Unity does not include test assemblies (NUnit, Unity TestRunner, and user script assemblies) when using the normal build pipeline, but does include them when using “Run on <Platform>” in the Test Runner Window.

Testing in Edit mode

In **Edit** mode, Unity runs tests from the Test Runner window.

Edit mode test **scripts** are defined by the file location you place them in. Valid locations:

Project Editor folder

Assembly Definition file that references test assemblies that are Editor-only

Precompiled assemblies that are in the Project's Editor folder

Click the **EditMode** button, then click **Create Test Script in current folder** to create a basic test script. Open and edit this in your preferred script editing software as required.

Note: When running in **Edit** mode, execute UnityTestAttribute in the EditorApplication.update callback loop.

Testing in Play mode

You need to place Play mode test scripts in a folder that an Assembly Definition file includes. The Assembly Definition file needs to reference test assemblies (Nunit and Unity TestRunner). Pre-defined Unity assemblies (such as *Assembly-CSharp.dll*) do not reference the defined assembly. This Assembly Definition file also needs to reference the assembly you want to test. This means that it's only possible to test code defined by other Assembly Definition files.

Unity does not include test assemblies in normal player builds; only when running through the Test Runner. If you need to test code in pre-defined assemblies, you can reference test assemblies from all the assemblies. However, you must manually remove these tests afterwards, so that Unity does not add them to the final player build.

To do this:

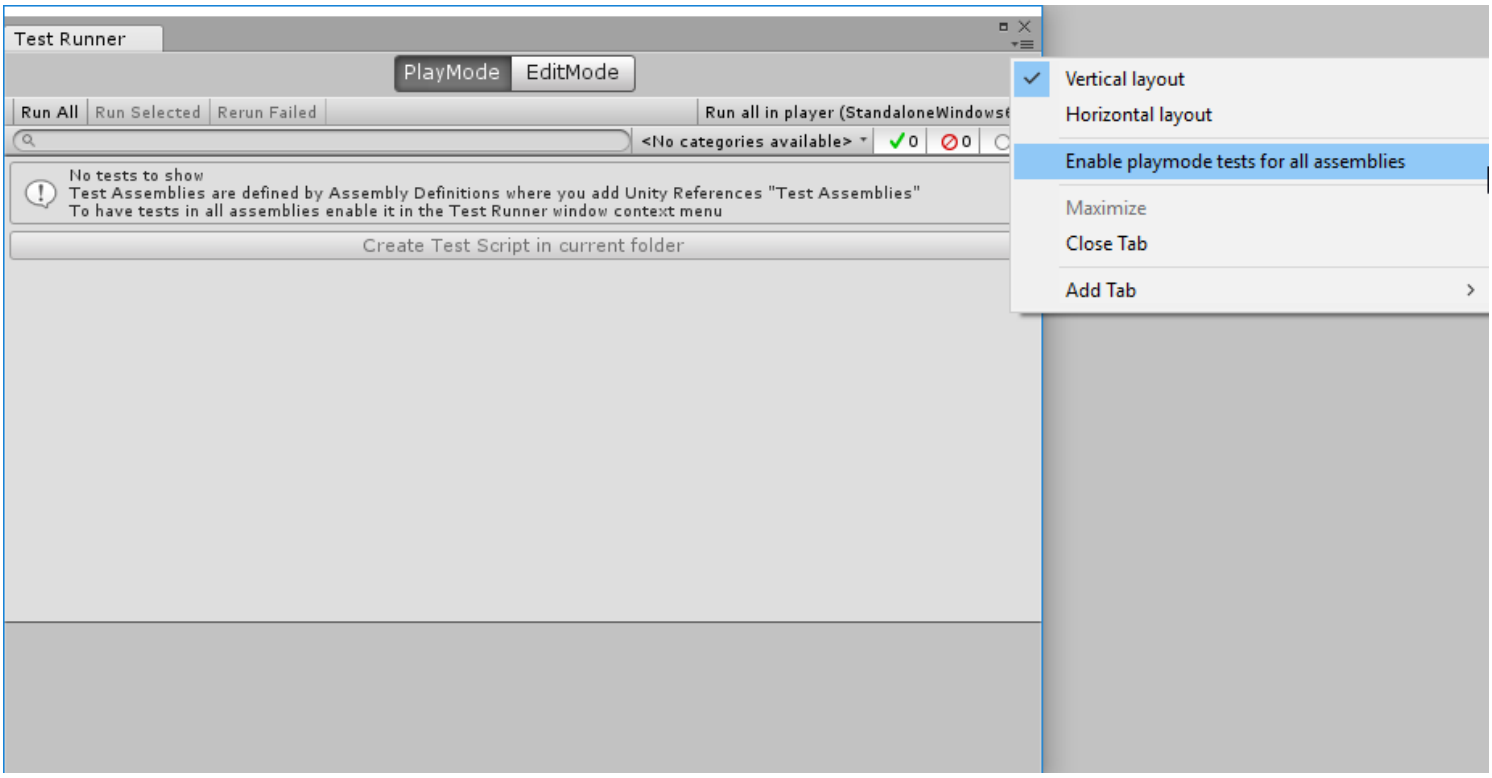
Save your project.

Go to **Window** > **General** > **Test Runner.**

Click the small drop-down menu in the top-right of the window.

Click **Enable playmode tests for all assemblies**.

In the dialog box appears, click **OK** to manually restart the Editor.



Note: Enabling **PlayMode** tests for all assemblies includes additional assemblies in your Project's build, which can increase your Project's size as well as build time.

To create **PlayMode** test scripts, select **PlayMode** in the Test Runner window and click **Create Test Script in current folder**. This button is greyed out if adding the script would result in a compilation error.

Note: Execute UnityTestAttribute as a coroutine when running in **Play** mode.

2018–03–21 Page amended with editorial review

Writing and executing tests in Unity Test Runner

Leave feedback

The Unity **Test Runner** tests your code in **Edit** mode and **Play** mode, as well as on target platforms such as Standalone, Android, or iOS.

The documentation on this page discusses writing and executing tests in the the Unity Test Runner, and assumes you understand both scripting and the Unity Test Runner.

Unity delivers test results in an XML format. For more information, see the NUnit documentation on XML format test results.

UnityTestAttribute

UnityTestAttribute requires you to return `IEnumerator`. In **Play** mode, execute the test as a coroutine. In **Edit** mode, you can yield `null` from the test, which skips the current frame.

Note: The **WebGL** and WSA platforms do not support `UnityTestAttribute`.

Regular NUnit test (works in Edit mode and Play mode)

```
[Test]
public void GameObject_CreatedWithGiven_WillHaveTheName()
{
    var go = new GameObject("MyGameObject");
    Assert.AreEqual("MyGameObject", go.name);
}
```

Example in Play mode

```
[UnityTest]
public IEnumerator GameObject_WithRigidBody_WillBeAffectedByPhysics()
{
    var go = new GameObject();
    go.AddComponent<Rigidbody>();
    var originalPosition = go.transform.position.y;

    yield return new WaitForFixedUpdate();

    Assert.AreNotEqual(originalPosition, go.transform.position.y);
}
```

Example in Edit mode:

```
[UnityTest]
public IEnumerator EditorUtility_WhenExecuted_ReturnsSuccess()
{
    var utility = RunEditorUtilityInTheBackgroud();

    while (utility.isRunning)
    {
        yield return null;
    }

    Assert.IsTrue(utility.isSuccess);
}
```

UnityPlatformAttribute

Use UnityPlatformAttribute to filter tests based on the the executing platform. It behaves like the NUnit PlatformAttribute.

```
[Test]
[UnityPlatform (RuntimePlatform.WindowsPlayer)]
public void TestMethod1()
{
   Assert.AreEqual(Application.platform, RuntimePlatform.WindowsPlayer);
}

[Test]
[UnityPlatform(exclude = new[] {RuntimePlatform.WindowsEditor })]
public void TestMethod2()
{
   Assert.AreNotEqual(Application.platform, RuntimePlatform.WindowsEditor);
}
```

To only execute Editor tests on a given platform, you can also use `UnityPlatform` .

PrebuildSetupAttriubte

Use PrebuildSetupAttribute if you need to perform any extra set-up before the test starts. To do this, specify the class type that implements the IPrebuildSetup interface. If you need to run the set-up code for the whole class

(for example, if you want to execute some code before the test starts, such as Asset preparation or set-up required for a specific test), implement the `IPrebuildSetup` interface in the class for tests.

```
public class TestsWithPrebuildStep : IPrebuildSetup
{
    public void Setup()
    {
        // Run this code before the tests are executed
    }

    [Test]
    //PrebuildSetupAttribute can be skipped because it's implemented in the same
    [PrebuildSetup(typeof(TestsWithPrebuildStep))]
    public void Test()
    {
        (...)
    }
}
```

Execute the `IPrebuildSetup` code before entering Play mode or building a player. Setup can use UnityEditor namespace and its function, but to avoid compilation errors, you must place it either in the *Editor* folder, or guard it with the `#if UNITY_EDITOR` directive.

LogAssert

A test fails if Unity logs a message other than a regular log or warning message. Use the LogAssert class to make a message expected in the log, so that the test does not fail when Unity logs that message.

A test also reports as failed if an expected message does not appear, or if Unity does not log any regular log or warning messages.

Example

```
[Test]
public void LogAssertExample()
{
    //Expect a regular log message
    LogAssert.Expect(LogType.Log, "Log message");
    //A log message is expected so without the following line
    //the test would fail
    Debug.Log("Log message");
    //An error log is printed
    Debug.LogError("Error message");
```

```
    //Without expecting an error log, the test would fail
    LogAssert.Expect(LogType.Error, "Error message");
}
```

MonoBehaviourTest

MonoBehaviourTest is a coroutine, and a helper for writing MonoBehaviour tests. Yield `MonoBehaviourTest` from the UnityTest attribute to instantiate the specified MonoBehaviour and wait for it to finish executing. Implement the IMonoBehaviourTest interface to indicate when the test is done.

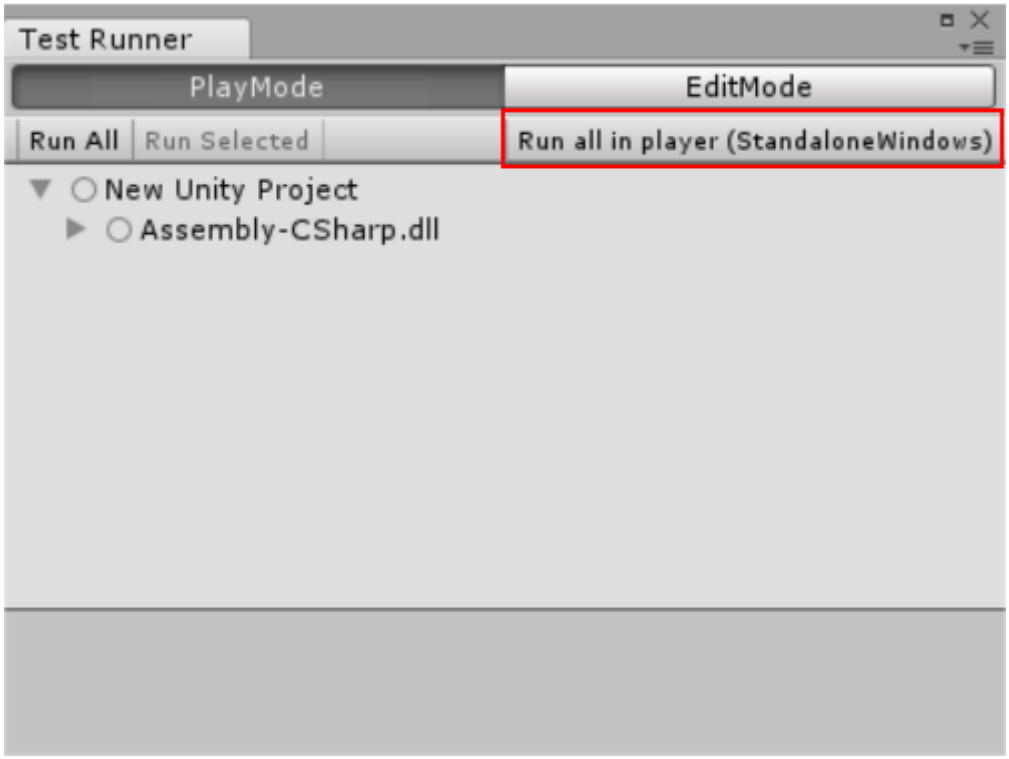
Example

```
[UnityTest]
public IEnumerator MonoBehaviourTest_Works()
{
    yield return new MonoBehaviourTest<MyMonoBehaviourTest>();
}

public class MyMonoBehaviourTest : MonoBehaviour, IMonoBehaviourTest
{
    private int frameCount;
    public bool IsTestFinished
    {
        get { return frameCount > 10; }
    }

     void Update()
     {
        frameCount++;
     }
}
```

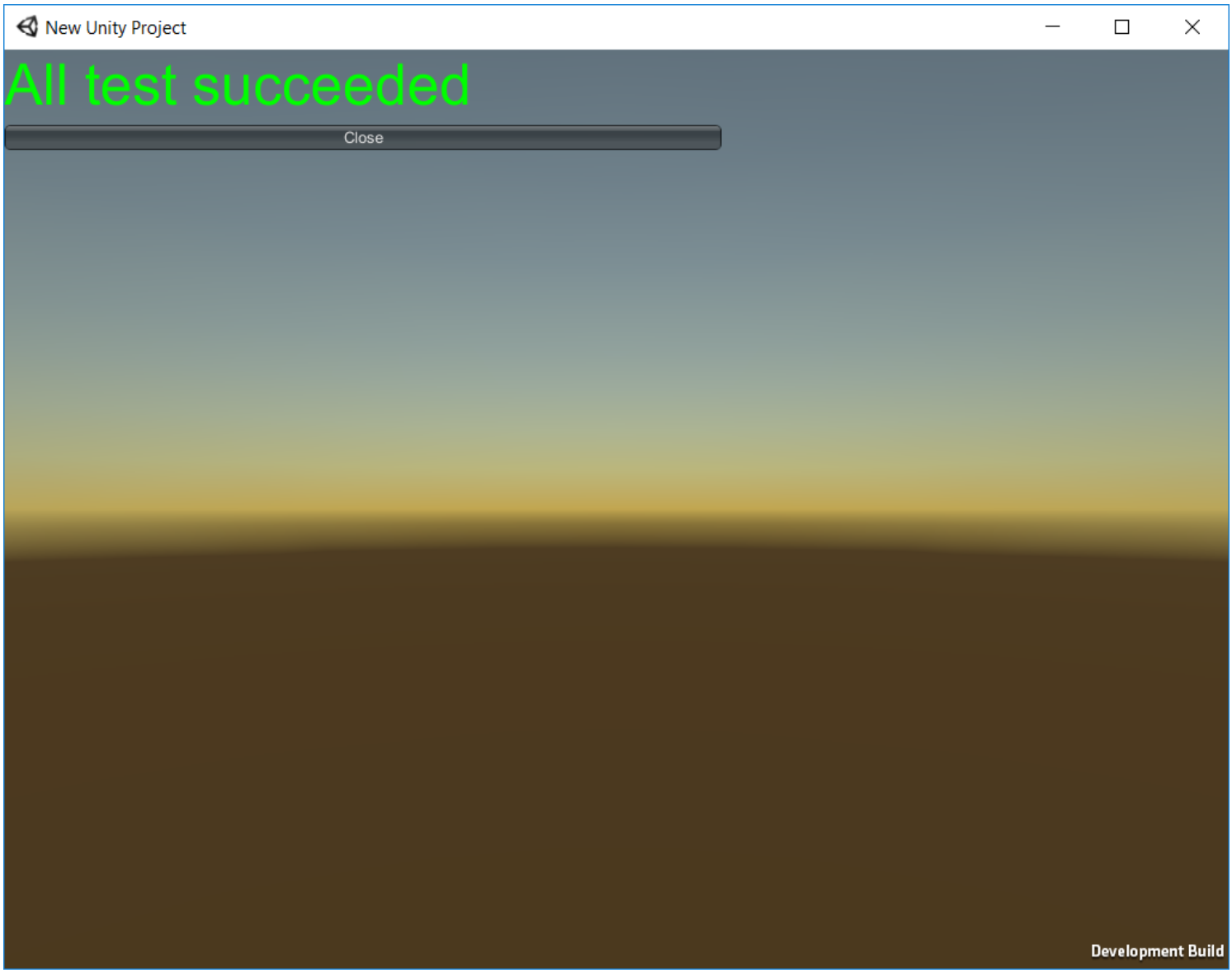
Running tests on platforms

In **Play** mode, you can run tests on specific platforms. The target platform is always the current Platform selected in Build Settings (menu: **File** > **Build Settings**). Click **Run all in the player** to build and run your tests on the currently active target platform.

Note that your current platform displays in brackets on the button. For example, in the screenshot below, the button reads **Run all in player (StandaloneWindows)**, because the current platform is Windows.



The test results display in the build once the test is complete.



To get the test results from the platform to the Editor running the test, both need to be on same network. The application running on the platform reports back the test results, displays the tests executed, and shuts down.

Note that some platforms do not support shutting down the application with Application.Quit. These continue running after reporting test results.

If Unity cannot instantiate the connection, you can visually see the tests succeed in the running application. Note that running tests on platforms with arguments, in this state, does not provide XML test results.

Running from the command line

To do this, run Unity with the following command line arguments:

`runTests` - Executes tests in the Project.

`testPlatform` - The platform you want to run tests on.

Available platforms:

`playmode` and `editmode`. Note that If unspecified, tests run in `editmode` by default.

Platform/Type convention is from the BuildTarget enum. The tested and official supported platforms:

StandaloneWindows

StandaloneWindows64

StandaloneOSXIntel

StandaloneOSXIntel64

iOS

tvOS

Android

PS4

XboxOne

`testResults` - The path indicating where Unity should save the result file. By default, Unity saves it in the Project's root folder.

Example

The following example shows a command line argument on Windows. The specific line may differ depending on your operating system.

```
>Unity.exe -runTests -projectPath PATH_TO_YOUR_PROJECT -testResults C:\temp\resu
```

Tip: On Windows, in order to read the result code of the executed command, run the following:

`start /WAIT Unity.exe ARGUMENT_LIST.`

Comparison utilities

The UnityEngine.TestTools.Utils namespace contains utility classes to compare Vector2, Vector3, Vector4, Quaternion, Color and `float` types using NUnit constraints.

2018–03–21 Page amended with editorial review

IL2CPP

Leave feedback

IL2CPP (Intermediate Language To C++) is a Unity-developed **scripting backend** which you can use as an alternative to Mono when building projects for various platforms. When building a project using **IL2CPP**, Unity converts IL code from **scripts** and assemblies to C++, before creating a native binary file (.exe, apk, .xap, for example) for your chosen platform. Some of the uses for IL2CPP include increasing the performance, security, and platform compatibility of your Unity projects.

For more information about using IL2CPP, refer to the The Unity IL2CPP blog series and the following Unity Manual pages.

Building a project using IL2CPP
How IL2CPP works
Optimising IL2CPP build times
See the Scripting Restrictions page for details about which platforms support IL2CPP.

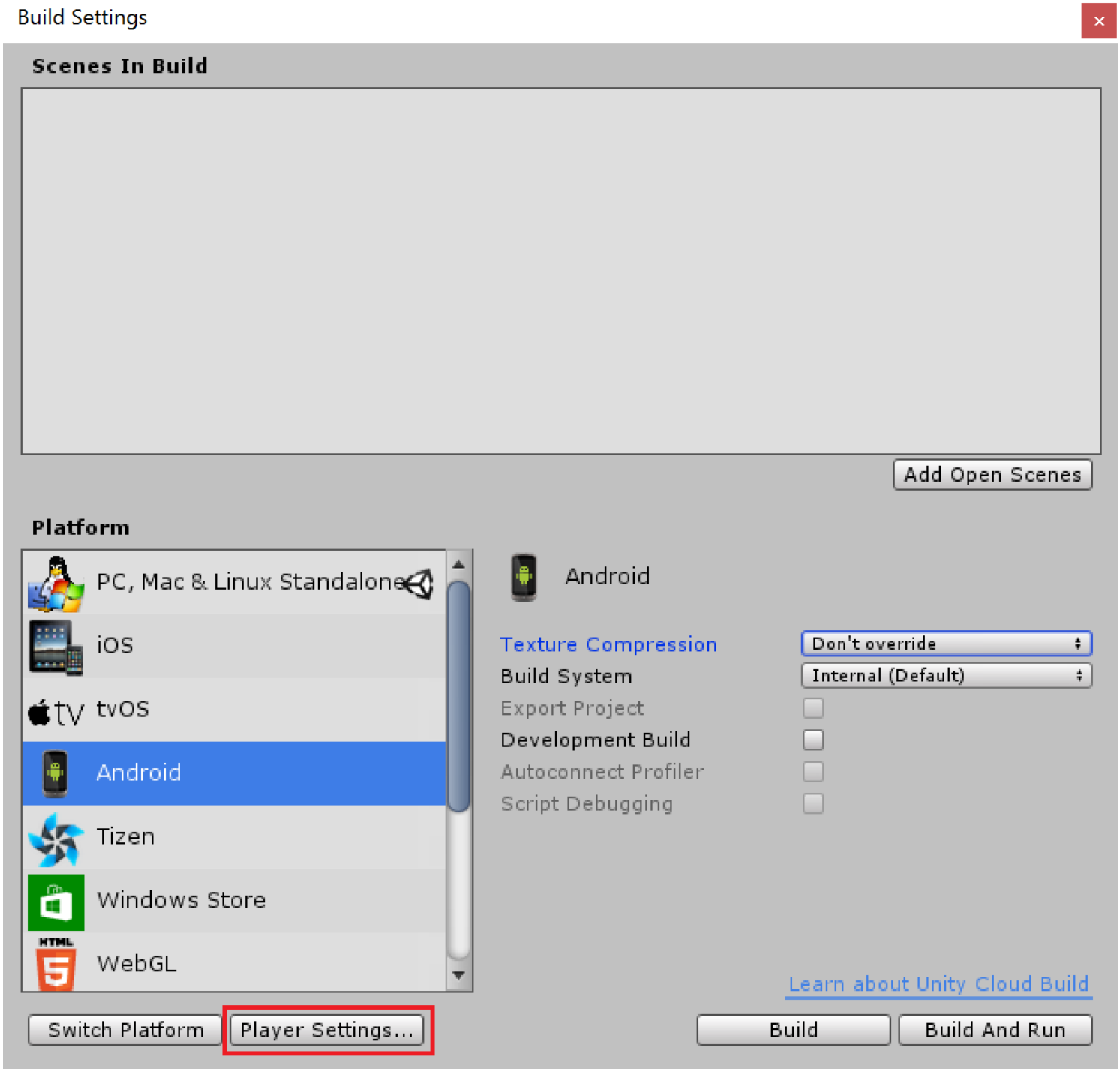
IL2CPP supports debugging of managed code in the same way as the Mono scripting backend.

• 2018–05–15 Page amended with limited editorial review

Building a project using IL2CPP

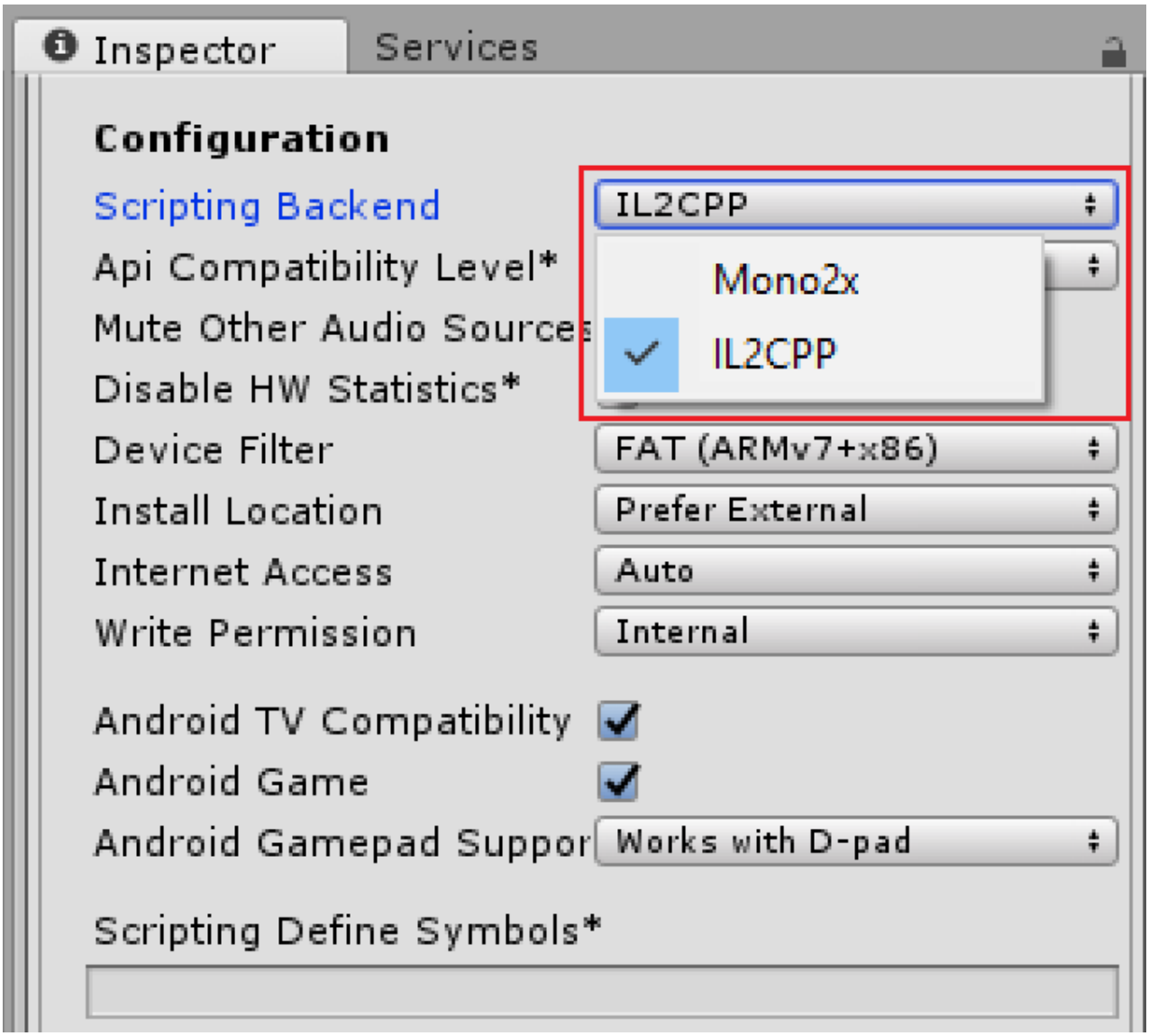
Leave feedback

To build your project using IL2CPP, open the Build Settings window (**File** > **Build Settings**). Select the platform you are building for, then click **Player Settings...** to open the PlayerSettings window in the Inspector.



The Build Settings window

In the PlayerSettings window for your target platform, scroll down to the **Configuration** section. For **Scripting Backend**, select **IL2CPP**.



The Configuration section of the PlayerSettings window

With IL2CPP selected as the Scripting back end, click **Build** in the Build Settings window. Unity begins the process of converting your C# code and assemblies into C++ before producing a binary file for your target platform.

Compiler options

Leave feedback

When using the IL2CPP **scripting backend**, it is possible to control how **il2cpp.exe** generates C++ code. Specifically, C# attributes can be used to enable or disable the following runtime checks listed below.

Option	*Description*	*Default*
Null checks	If this option is enabled, C++ code generated by IL2CPP will contain null checks and throw managed NullReferenceException exceptions as necessary. If this option is disabled, the null checks are not be emitted into the generated C++ code. For some projects, disabling this option may improve runtime performance. However, any access to null values in the generated code will not be protected, and may lead to incorrect behavior. *Usually* the game will crash soon after the null value is dereferenced, but we cannot guarantee this. Disable this option with caution.	Enabled
Array bounds checks	If this option is enabled, C++ code generated by IL2CPP will contain array bounds checks and throw managed IndexOutOfRangeException exceptions as necessary. If this option is disabled, the array bounds checks will not be emitted into the generated C++ code. For some projects, disabling this option may improve runtime performance. However, any access to an array with invalid indices in the generated code will not be protected, and may lead to incorrect behavior, including reading from or writing to arbitrary memory locations. In most cases, these memory accesses will occur without any immediate side effects, and may silently corrupt the state of the game. Disable this option with extreme caution.	Enabled
Divide by zero checks	If this option is enabled, C++ code generated by IL2CPP will contain divide by zero checks for integer division and throw managed DivideByZeroException exceptions as necessary. If this option is disabled, the divide by zero checks on integer division will not be emitted into the generated C++ code. For most projects this option should be disabled. Enable it only if divide by zero checks are required, as these checks have a runtime cost.	Disable

The runtime checks can be enabled or disabled in C# code using the `Il2CppSetOptions` attribute. To use this attribute, find the *Il2CppSetOptionsAttribute.cs* source file in the *IL2CPP* directory in the Unity Editor installation on your computer. (*Data\il2cpp* on Windows, *Contents/Frameworks/il2cpp* on OS X). Copy this source file into the ***Assets*** folder in your project.

Now use the attribute as in the example below.

```
[Il2CppSetOption(Option.NullChecks, false)]
public static string MethodWithNullChecksDisabled()
{
    var tmp = new object();
    return tmp.ToString();
}
```

You can apply `Il2CppSetOptions` attribute to types, methods, and properties. Unity uses the attribute from the most local scope.

```
[Il2CppSetOption(Option.NullChecks, false)]
public class TypeWithNullChecksDisabled
{
    public static string AnyMethod()
    {
        // Null checks will be disabled in this method.
        var tmp = new object();
        return tmp.ToString();
    }

    [Il2CppSetOption(Option.NullChecks, true)]
```

```
    public static string MethodWithNullChecksEnabled()
    {
        // Null checks will be enabled in this method.
        var tmp = new object();
        return tmp.ToString();
    }
}
```

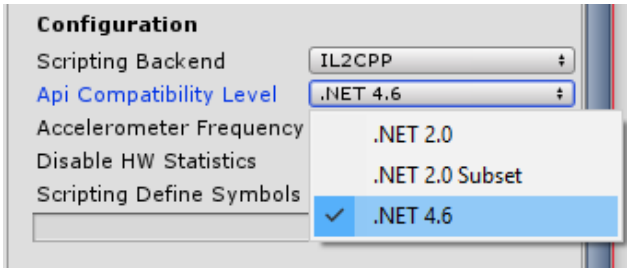
```
public class SomeType
{
    [Il2CppSetOption(Option.NullChecks, false)]
    public string PropertyWithNullChecksDisabled
    {
        get
        {
            // Null checks will be disabled here.
            var tmp = new object();
            return tmp.ToString();
        }
        set
        {
            // Null checks will be disabled here.
            value.ToString();
        }
    }

    public string PropertyWithNullChecksDisabledOnGetterOnly
    {
        [Il2CppSetOption(Option.NullChecks, false)]
        get
        {
            // Null checks will be disabled here.
            var tmp = new object();
            return tmp.ToString();
        }
        set
        {
            // Null checks will be enabled here.
            value.ToString();
        }
    }
}
```

Windows Runtime support

Leave feedback

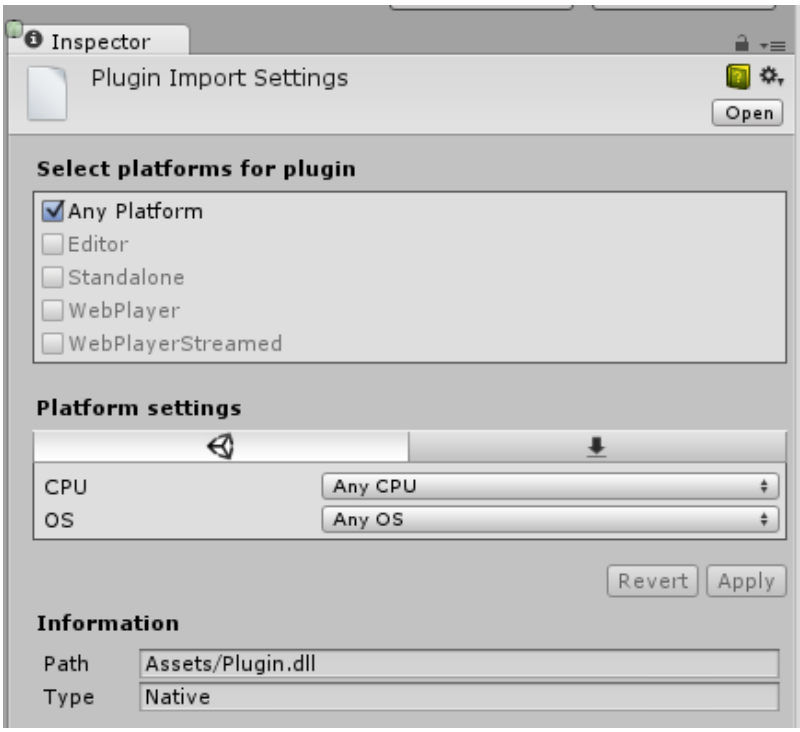
Unity includes Windows Runtime support for IL2CPP on **Universal Windows Platform** and **Xbox One** platforms. Use Windows Runtime support to call into both native system Windows Runtime APIs as well as custom .winmd files directly from managed code (scripts and DLLs).

To automatically enable Windows Runtime support in IL2CPP, go to PlayerSettings (**Edit** > **Project Settings** > **Player**), navigate to the **Configuration** section, and set the **Api Compatibility Level** to **.NET 4.6**.



The **Configuration** section of the PlayerSettings window. The options shown above change depending on your chosen build platform.

Unity automatically references Windows Runtime APIs (such as *Windows.winmd* on Universal Windows Platform) when it has Windows Runtime support enabled. To use custom .winmd files, import them (together with any accompanying DLLs) into your Unity project folder. Then use the Plugin Inspector to configure the files for your target platform.



Use the Plugin Inspector to configure custom .winmd files for specific platforms

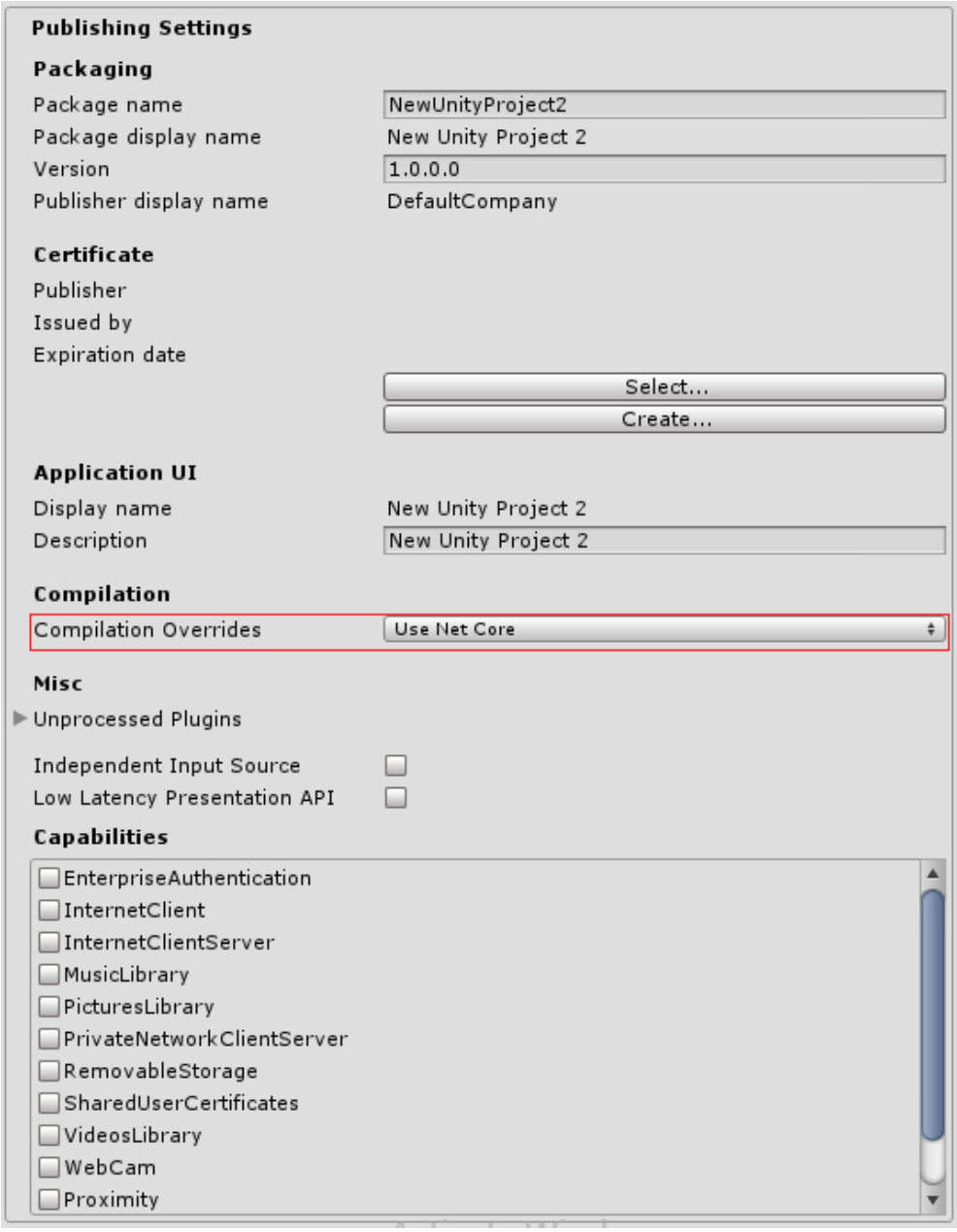
In your Unity project's scripts you can use the `ENABLE_WINMD_SUPPORT` #define directive to check that your project has Windows Runtime support enabled. Use this before a call to .winmd Windows APIs or custom .winmd scripts to ensure they can run and to ensure any scripts not relevant to Windows ignore them. Note, this is only supported in C# scripts. See the examples below.

Examples

C#

```
void Start() {
  #if ENABLE_WINMD_SUPPORT
    Debug.Log("Windows Runtime Support enabled");
    // Put calls to your custom .winmd API here
  #endif
}
```

In addition to being defined when Windows Runtime support is enabled in IL2CPP, it is also defined in .NET when you set **Compilation Overrides** to **Use Net Core**.



The Publishing Settings section of the PlayerSettings Inspector window, with **Compilation Overrides** highlighted in red

• 2017–05–16 Page amended with no editorial review

How IL2CPP works

Leave feedback

Upon starting a build using IL2CPP, Unity automatically performs the following steps:

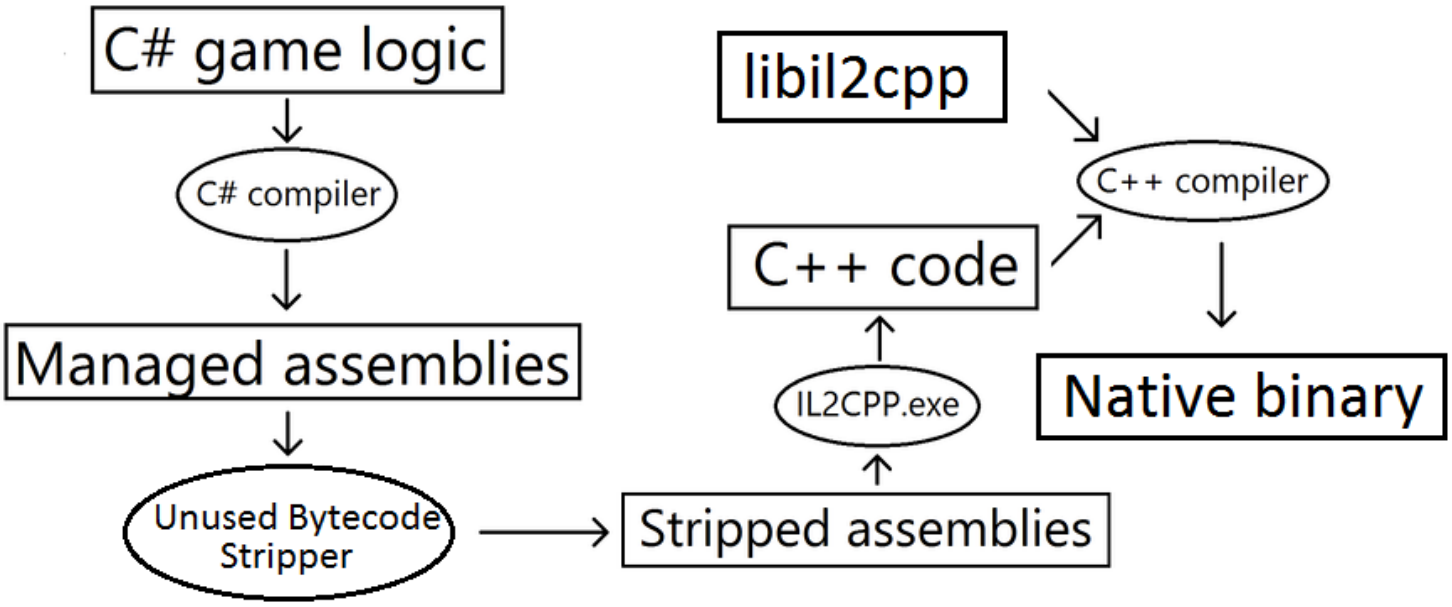
Unity Scripting API code is compiled to regular .NET DLLs (managed assemblies).

All managed assemblies that aren't part of **scripts** (such as plugins and base class libraries) are processed by a Unity tool called Unused Bytecode Stripper, which finds all unused classes and methods and removes them from these DLLs (Dynamic Link Library). This step significantly reduces the size of a built game.

All managed assemblies are then converted to standard C++ code.

The generated C++ code and the runtime part of IL2CPP is compiled using a native platform compiler.

Finally, the code is linked into either an executable file or a DLL, depending on the platform you are targeting.



A diagram of the automatic steps taken when building a project using IL2CPP

IL2CPP provides a few useful options which you can control by attributes in your scripts. See documentation on Platform-dependent compilation for further information.

Optimizing IL2CPP build times

Leave feedback

Project build times can be much longer when building a project with IL2CPP. However, there are several ways to reduce the build time significantly:

Use incremental building

When using incremental building, the C++ compiler only recompiles files that have changed since the last build. To use incremental building, build your project to a previous build location (without deleting the target directory).

Exclude project and target build folders from anti-malware software scans

You can improve build times by disabling anti-malware software before building your project. (Testing by Unity Technologies found that build times decreased by 50 – 66% after disabling Windows Defender on a fresh Windows 10 installation.)

Store your project and target build folder on a Solid State Drive (SSD)

Solid State Drives (SSDs) have faster read/write speed, when compared to traditional Hard Disk Drives (HDD). Converting IL code to C++ and compiling it involves a large number of read/write operations. A faster storage device speeds up this process.

Managed bytecode stripping with IL2CPP

Leave feedback

Managed bytecode stripping removes unused code from managed assemblies (DLLs). The process works by defining root assemblies, then using static code analysis to determine what other managed code those root assemblies use. Any code that is not reachable is removed. Bytecode stripping will *not* obfuscate code, nor will it modify code that is used in any way.

For a given Unity player build, the root assemblies are those compiled by the Unity Editor from script code (for example, *Assembly-CSharp.dll*). Any assemblies compiled from script code will not be stripped, but other assemblies will. This includes:

Assemblies you add to a project
Unity Engine assemblies
.NET class library assemblies (e.g mscorlib.dll, System.dll)

Managed bytecode stripping is always enabled when the IL2CPP **scripting backend** is used. In this case, the Stripping Level option is replaced with a Boolean option named Strip Engine Code. If this option is enabled, unused modules and classes in the *native* Unity Engine code will be also removed. If it is disabled, all of the modules and classes in the native Unity Engine code will be preserved.

The *link.xml* file (described below) can be used to effectively disable bytecode stripping by preserving both types and full assemblies. For example, to prevent the System assembly from being stripped, the following link.xml file can be used:

```
<linker>
       <assembly fullname="System" preserve="all"/>
</linker>
```

Tips

How to deal with stripping when using reflection

Stripping depends highly on static code analysis and sometimes this can't be done effectively, especially when dynamic features like reflection are used. In such cases, it is necessary to give some hints as to which classes shouldn't be touched. Unity supports a per-project custom stripping blacklist. Using the blacklist is a simple matter of creating a link.xml file and placing it into the **Assets** folder (or any subdirectory of Assets). An example of the contents of the link.xml file follows. Classes marked for preservation will not be affected by stripping:

```
<linker>
       <assembly fullname="System.Web.Services">
               <type fullname="System.Web.Services.Protocols.SoapTypeStubInfo" p
       </assembly>
```

```
        <assembly fullname="System">
                <type fullname="System.Net.Configuration.WebRequestModuleHandler"
                <type fullname="System.Net.HttpRequestCreator" preserve="all"/>
                <type fullname="System.Net.FileWebRequestCreator" preserve="all"/
        </assembly>

        <assembly fullname="mscorlib">
                <type fullname="System.AppDomain" preserve="fields"/>
                <type fullname="System.InvalidOperationException" preserve="field
                        <method signature="System.Void .ctor()"/>
                </type>
                <type fullname="System.Object" preserve="nothing">
                       <method name="Finalize"/>
                </type>
        </assembly>
 </linker>
```

A project can include multiple link.xml files. Each link.xml file can specify a number of different options. The assembly element indicates the managed assembly where the nested directives should apply.

The type element is used to indicate how a specific type should be handled. It must be a child of the assembly element. The fullname attribute can accept the '*' wild card to match one or more characters.

The preserve attribute can take on one of three values:

all: Keep everything from the given type (or assembly, for IL2CPP only).
fields: Keep only the fields of the given type.
nothing: Keep only the given type, but none of its contents.
The method element is used to indicate that a specific method should be preserved. It must be a child of the type element. The method can be specified by name or by signature.

In addition to the link.xml file, the C# `[Preserve]` attribute can be used in source code to prevent the linker from stripping that code. This attribute behaves slightly differently than corresponding entries in a link.xml file:

Assembly: preserves all types in the assembly (as if a `[Preserve]` attribute were on each type)
Type: preserves the type and its default constructor
Method: preserves the method's declaring type, return type, and the types of all of its arguments
Property, Field, Event: preserves the declaring type and return type of the property, field, or event
The stripped assemblies are output to a directory below the Temp directory in the project (the exact location varies depending on the target platform). The original, unstripped assemblies are available in the not-stripped directory in the same location as the stripped assemblies. A tool like ILSpy can be used to inspect the stripped and unstripped assemblies to determine what parts of the code were removed.

2017–09–01 Page amended with limited editorial review

2017–05–26 - Documentation-only update in Unity User Manual for Unity 5.6

2017–09–01 - Added advice on using C# `[Preserve]` attribute for Unity 2017.1

Integrated development environment (IDE) support

Leave feedback

An integrated development environment (IDE) is a piece of computer software that provides tools and facilities to make it easier to develop other pieces of software. Unity supports the following IDEs:

Visual Studio (default IDE on Windows and macOS)

Unity installs Visual Studio by default when you install Unity on Windows and macOS. On Windows, you can choose to exclude it when you select which components to download and install. Visual Studio is set as the **External Script Editor** in the Preferences (menu: **Unity > Preferences > External Tools > External Script Editor**). With this option enabled, Unity launches Visual Studio and uses it as the default editor for all script files.

On macOS, Unity includes Visual Studio for Mac as the C# IDE. Visual Studio Tools for Unity (VSTU) provides Unity integration for Visual Studio for Mac (VS4M). For information on setting up and using Visual Studio for Mac, see the following Microsoft documentation pages:

Visual Studio Tools for Unity
Setup Visual Studio for Mac Tools for Unity
Using Visual Studio for Mac Tools for Unity
On Windows, Unity also includes Visual Studio 2017 Community.

Visual Studio Code (Windows, macOS, Linux)

Unity supports opening scripts in Visual Studio Code (VS Code). To open scripts in VS Code, select it as the **External Script Editor** in the Editor Preferences (menu: **Unity > Preferences > External Tools > External Script Editor**). For information on using VS Code with Unity, see Visual Studio's documentation on Unity Development with VS Code.

Prerequisites

To use Visual Studio Code for C# code editing and Unity C# debugging support, you need to install:

Mono (only required on macOS and Linux)
Visual Studio C# Extension
Visual Studio Unity Debugger Extension (does not support debugging on .NET Framework 4.6)

JetBrains Rider (Windows, macOS, Linux)

Unity supports opening scripts in JetBrains Rider. To open scripts in Rider, select it as the **External Script Editor** in the Editor Preferences (menu: **Unity > Preferences > External Tools > External Script Editor**).

Rider is built on top of ReSharper and includes most of its features. It supports all of C# 6.0's features as well as C# debugging on the .NET 4.6 scripting runtime in Unity. For more information, see JetBrains documentation on Rider for Unity.

2018–07–03 Page published with editorial review

Debugging C# code in Unity

Leave feedback

Using a debugger allows you to inspect your source code while your application or game is running. Unity supports debugging of C# code using the following code editors:

Visual Studio (with the Visual Studio Tools for Unity plug-in)

Visual Studio for Mac

Jetbrains Rider

Visual Studio Code

Although these code editors vary slightly in the debugger features they support, all provide basic functionality like break points, single stepping, and variable inspection.

Managed code debugging in Unity works on all platforms except **WebGL**. It works with both the Mono and **IL2CPP scripting backends**.

Configuring the code editor

Visual Studio (Windows)

The Unity Editor installer includes an option to install Visual Studio with the Visual Studio Tools for Unity plug-in. This is the recommended way to set up Visual Studio for debugging with Unity.

If Visual Studio is already installed on your computer, use its **Tools** > **Extensions and Updates** menu to locate and install the Visual Studio Tools for Unity plug-in.

Visual Studio for Mac

The Unity Editor installer includes an option to install Visual Studio for Mac. This is the recommended way to set up Visual Studio for Mac for debugging with Unity.

If Visual Studio for Mac is already installed on your computer, use its Extension Manager to locate and install the Visual Studio Tools for Unity plug-in.

JetBrains Rider

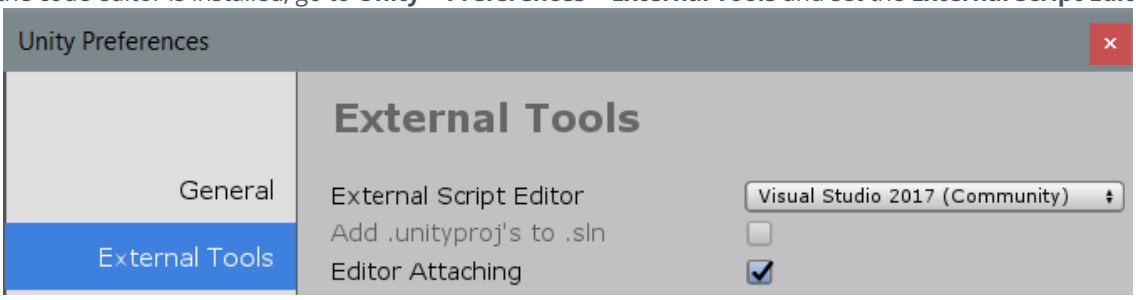
The default installation of JetBrains Rider can debug code in Unity on Windows or Mac. Please visit the JetBrains website to install it.

VS Code

VS Code requires you to install an extension to debug code in Unity. Please follow the instructions specific to this extension to install it.

Unity Preferences

When the code editor is installed, go to **Unity > Preferences > External Tools** and set the **External Script Editor** to your chosen code editor.



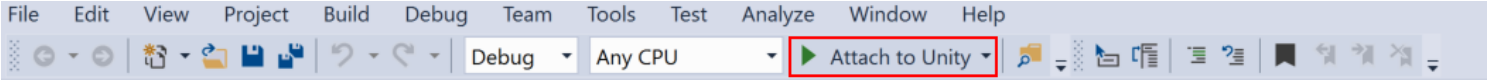
Debugging in the Editor

You can debug script code running in the Unity Editor when the Unity Editor is in Play Mode. Before attempting to debug, ensure the **Editor Attaching** option is enabled in the Unity Preferences. This option causes the Editor to use Just In Time (JIT) compilation to execute managed code with debugging information.

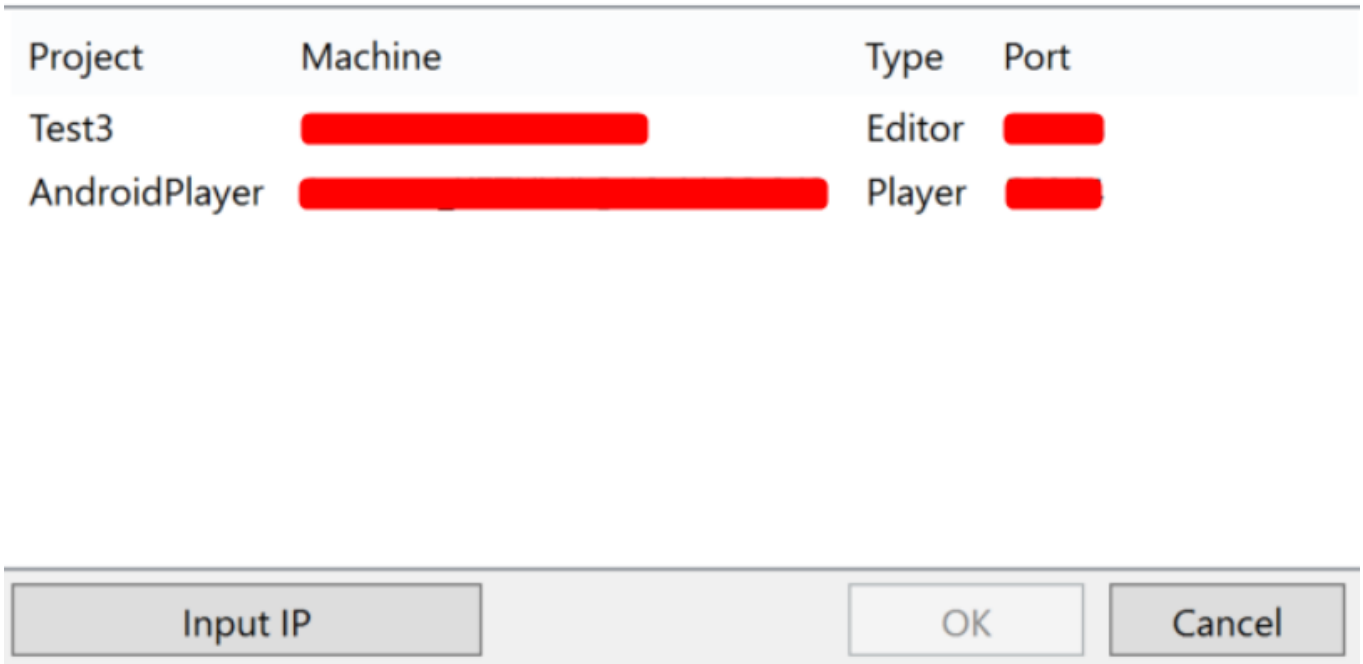
First, set a breakpoint in the code editor on a line of script code where the debugger should stop. In Visual Studio for example, click on the column to the left of your code, on the line you want to stop the debugger (as shown below). A red circle appears next to the line number and the line is highlighted.

```
5   public class ExampleScript : MonoBehaviour
6   {
7       // Use this for initialization
8       void Start ()
9       {
10          Debug.Log("Debug message here");
11      }
12  }
```

Next, attach the code editor to the Unity Editor. This option varies depending on the code editor, and is often a different option from the code editor's normal debugging process. In Visual Studio, the option looks like this:



Some code editors may allow you to select an instance of Unity to debug. For example, in Visual Studio, the **Debug > Attach Unity Debugger** option exposes this capability.



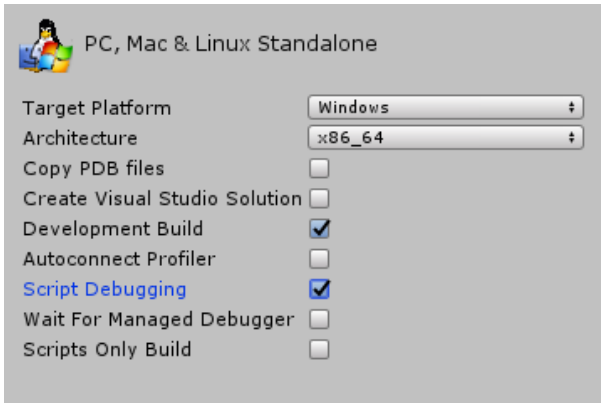
When you have attached the code editor to the Unity Editor, return to the Unity Editor and enter Play Mode. When the code at the breakpoint is executed, the debugger will stop, for example:

```
5   public class ExampleScript : MonoBehaviour
6   {
7       // Use this for initialization
8       void Start ()
9       {
10          Debug.Log("Debug message here");
11      }
12  }
```

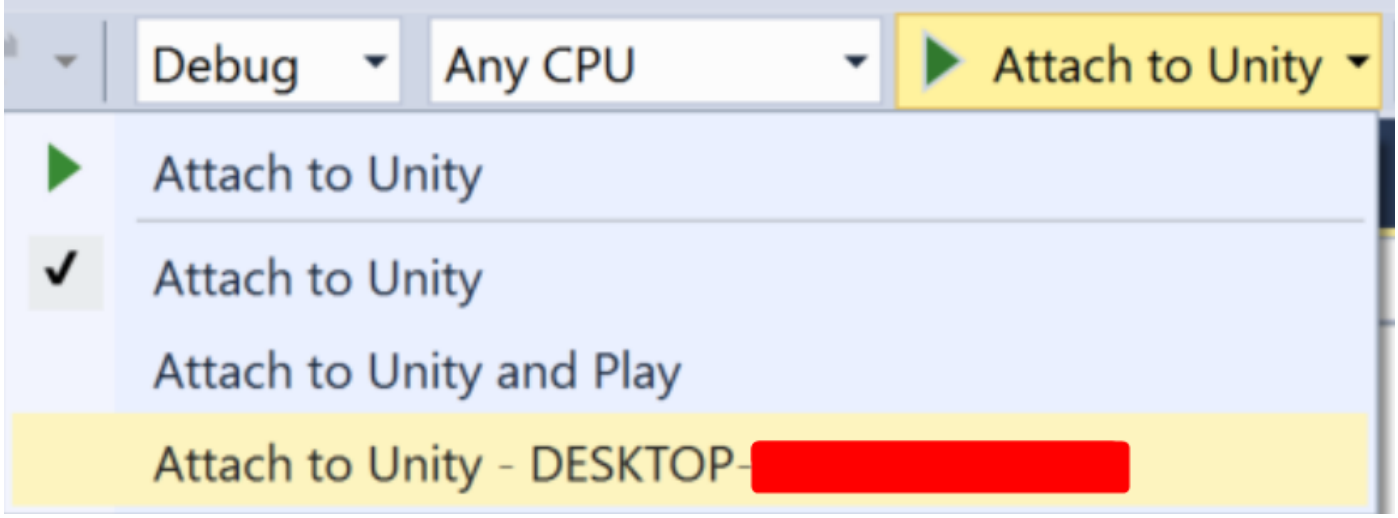
While the code editor is at a breakpoint, you can view the contents of variables step by step. The Unity Editor will be unresponsive until you choose the continue option in the debugger, or stop debugging mode.

Debugging in the Player

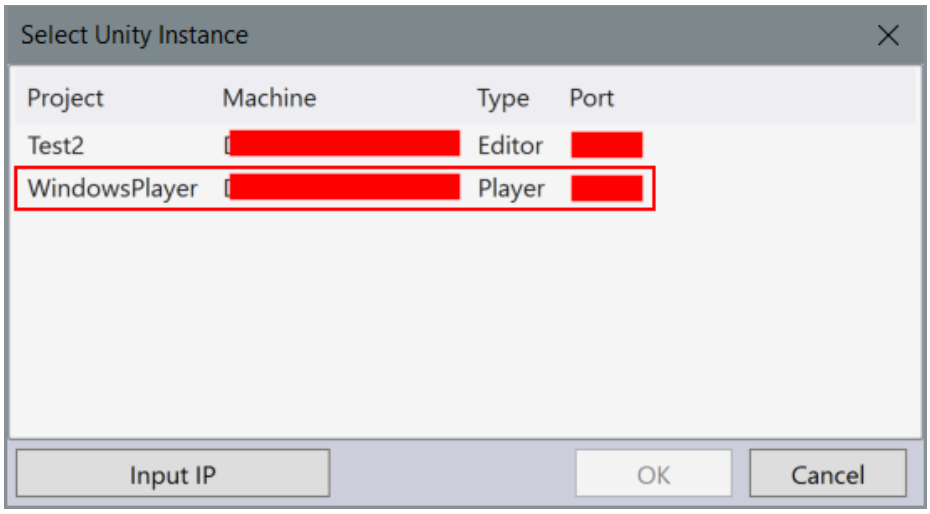
To debug script code running in a Unity Player, ensure that you enable the “Development Build” and ”Script Debugging” options before you build the Player (these options are located in **File > Build Settings**). Enable the “Wait For Managed Debugger” option to make the Player wait for a debugger to be attached before it executes any script code.



To attach the code editor to the Unity Player, select the IP address (or machine name) and port of your Player. In Visual Studio, the drop-down menu on the “Attach To Unity” option looks like this:



The **Debug** > **Attach Unity Debugger** option looks like this:

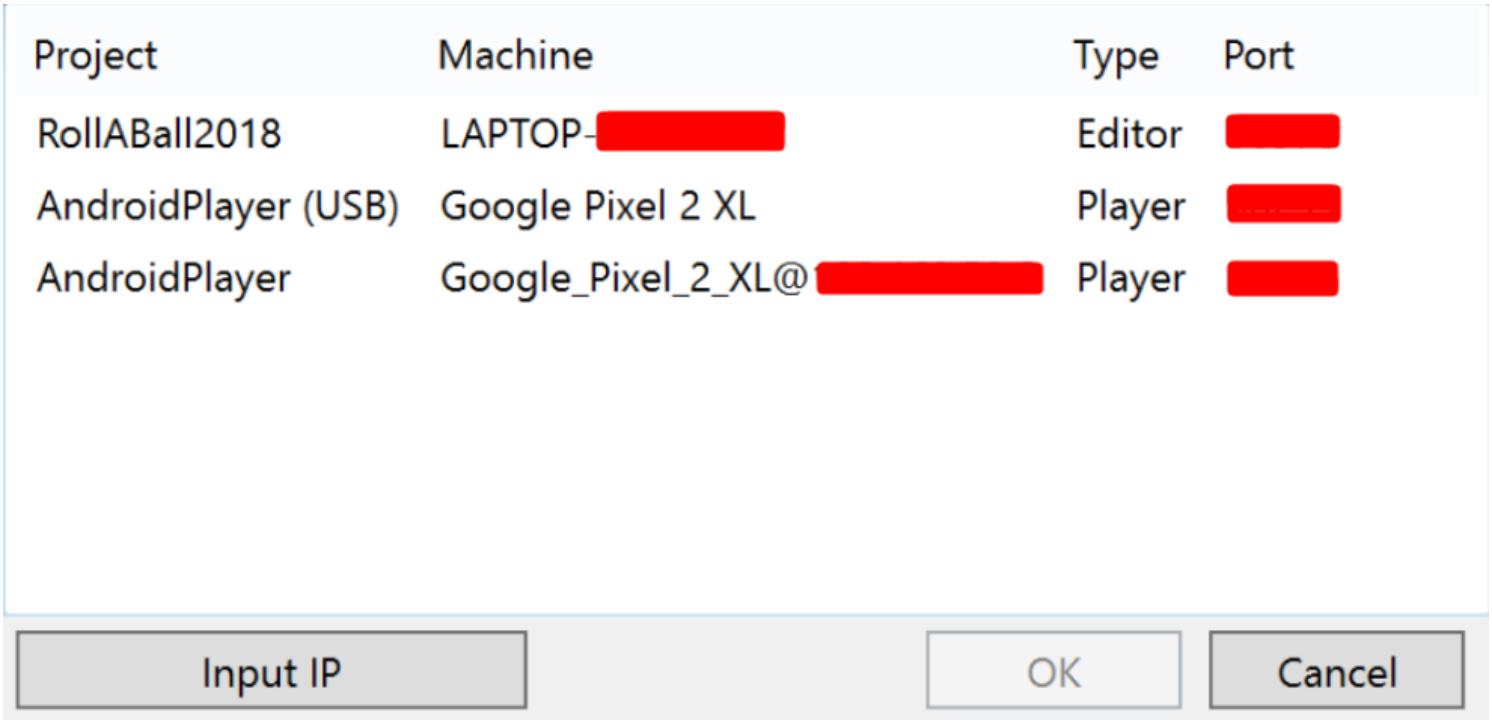


Make sure you attach the debugger to the Player, and not to the Unity Editor (if both are running). When you have attached the debugger, you can proceed with debugging normally.

Debugging on Android and iOS devices

Android

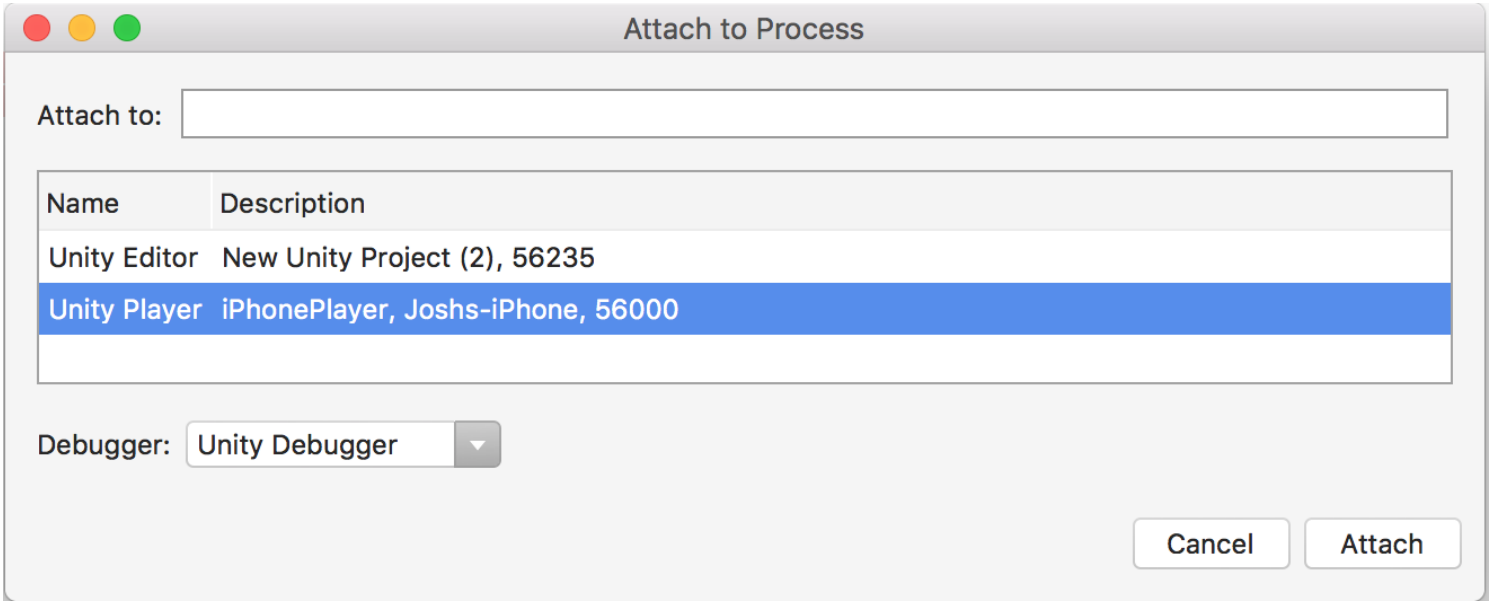
When debugging a Player running on an Android device, connect to the device via USB or TCP. For example, to connect to an Android device in Visual Studio (Windows), select **Debug > Attach Unity Debugger** option. A list of devices running a Player instance will appear:



In this case, the phone is connected via USB and Wi-Fi on the same network as the workstation running the Unity Editor and Visual Studio.

iOS

When debugging a Player running on an iOS device, connect to the device via TCP. For example, to connect to an Android device in Visual Studio (Mac), select **Debug** > **Attach Unity Debugger** option. A list of devices running a Player instance will appear:



Ensure that the device only has one active network interface (for example, Wi-Fi or cellular data) and that there is no firewall between the IDE and the device blocking the TCP port (port number 56000 in the above screenshot).

Troubleshooting the debugger

Most problems with the debugger occur because the code editor is unable to locate the Unity Editor or the Player. This means that it can't attach the debugger properly. Because the debugger uses a TCP connection to the Editor or Player, connection issues are often

caused by the network. Here are a few steps you can take to troubleshoot basic connection issues.

Ensure you attach the debugger to the correct Unity instance

You can attach code editor to any Unity Editor or Unity Player on the local network that has debugging enabled. When attaching the debugger to ensure that you are attaching to the correct instance. If you know the IP address or machine name of the device on which you are running the Unity Player, this helps to locate the correct instance.

Verify the network connection to the Unity instance

The code editor uses the same mechanism to locate a Unity instance to debug as the Unity **Profiler** uses. If the code editor cannot find the Unity instance you expect it to find, try to attach the Unity **Profiler** to that instance. If the Unity **Profiler** cannot find it either, a firewall might exist on the machine which you are running the code editor on or the machine which you are running the Unity instance on (or possibly both). If a firewall is in place, see the information about firewall settings below.

Ensure the device has only one active network interface

Many devices have multiple network interfaces. For example, a mobile phone may have both an active cellular connection and an active Wi-Fi connection. To properly connect the debugger for TCP, the IDE needs to make a network connection to the correct interface on the device. If you plan to debug via Wi-Fi, for example, make sure you put the device in airplane mode to disable all other interfaces, then enable Wi-Fi.

You can determine which IP address the Unity Player is telling the IDE to use by looking in the **Player Log**. Look for a part of the log like this:

```
Multi-casting "[IP] 10.0.1.152 [Port] 55000 [Flags] 3 [Guid] 2575380029 [EditorId] 4264788666 [V
```

This message indicates the IDE will try to use the IP address 10.0.1.152 and port 56000 to connect to the device. This IP address and port must be reachable from the computer running the IDE.

Check the firewall settings

The Unity instance communicates with the code editor via a TCP connection. On most Unity platforms, this TCP connection occurs on an arbitrarily chosen port. Normally, you should not need to know this port, as the code editor should detect it automatically. If that does not work, try to use a network analysis tool to determine which ports might be blocked either on the machine where you are running the code editor, or the machine or device where you are running the Unity instance. When you find the ports, make sure that your firewall allows access to both the port on the machine running the code editor, and the machine running the Unity instance.

Verify the managed debugging information is available

If the debugger does attach, but breakpoints don't load, the debugger may not be able to locate the managed debugging information for the code. Managed code debugging information is stored in files named .dll.mdb or .pdb, next to the managed assembly (.dll file) on disk.

When the correct preferences and build options are enabled (see above, Unity will generate this debugging information automatically. However, Unity cannot generate this debugging information for managed plugins in the Project. It is possible to debug code from managed plugins if the associated .dll.mdb or .pdb files are next to the managed plugins in the Unity project on disk.

2018–09–06 Page published with editorial review

Managed Code Debugging added in 2018.2

Event System

Leave feedback

The **Event System** is a way of sending events to objects in the application based on input, be it keyboard, mouse, touch, or custom input. The **Event System** consists of a few **components** that work together to send events.

Overview

When you add an Event System component to a **GameObject** you will notice that it does not have much functionality exposed, this is because the Event System itself is designed as a manager and facilitator of communication between Event System modules.

The primary roles of the Event System are as follows:

Manage which GameObject is considered selected
Manage which Input Module is in use
Manage Raycasting (if required)
Updating all Input Modules as required

Input Modules

An input module is where the main logic of how you want the Event System to behave lives, they are used for

Handling Input
Managing event state
Sending events to **scene** objects.

Only one Input Module can be active in the Event System at a time, and they must be components on the same GameObject as the Event System component.

If you wish to write a custom input module it is recommended that you send events supported by existing **UI** components in Unity, but you are also able to extend and write your own events as detailed in the Messaging System documentation.

Raycasters

Raycasters are used for figuring out what the pointer is over. It is common for Input Modules to use the Raycasters configured in the scene to calculate what the pointing device is over.

There are 3 provided Raycasters that exist by default:

Graphic Raycaster - Used for UI elements
Physics 2D Raycaster - Used for 2D physics elements
Physics Raycaster - Used for 3D physics elements

If you have a 2d / 3d Raycaster configured in your scene it is easily possible to have non UI elements receive messages from the Input Module. Simply attach a script that implements one of the event interfaces.

Messaging System

Leave feedback

The new **UI** system uses a messaging system designed to replace SendMessage. The system is pure C# and aims to address some of the issues present with SendMessage. The system works using custom interfaces that can be implemented on a MonoBehaviour to indicate that the **component** is capable of receiving a callback from the messaging system. When the call is made a target GameObject is specified; the call will be issued on all components of the GameObject that implement the specified interface that the call is to be issued against. The messaging system allows for custom user data to be passed, as well as how far through the GameObject hierarchy the event should propagate; that is should it just execute for the specified GameObject, or should it also execute on children and parents. In addition to this the messaging framework provides helper functions to search for and find **GameObjects** that implement a given messaging interface.

The messaging system is generic and designed for use not just by the UI system but also by general game code. It is relatively trivial to add custom messaging events and they will work using the same framework that the UI system uses for all event handling.

How Do I Define A Custom Message?

If you wish to define a custom message it is relatively simple. In the UnityEngine.EventSystems namespace there is a base interface called 'IEventSystemHandler'. Anything that extends from this can be considered as a target for receiving events via the messaging system.

```
public interface ICustomMessageTarget : IEventSystemHandler
{
    // functions that can be called via the messaging system
    void Message1();
    void Message2();
}
```

Once this interface is defined then it can be implemented by a MonoBehaviour. When implemented it defines the functions that will be executed if the given message is issued against this MonoBehaviours GameObject.

```
public class CustomMessageTarget : MonoBehaviour, ICustomMessageTarget
{
    public void Message1()
    {
        Debug.Log ("Message 1 received");
    }

    public void Message2()
    {
        Debug.Log ("Message 2 received");
```

```
        }
    }
```

Now that a script exists that can receive the message we need to issue the message. Normally this would be in response to some loosely coupled event that occurs. For example, in the UI system we issue events for such things as PointerEnter and PointerExit, as well as a variety of other things that can happen in response to user input into the application.

To send a message a static helper class exists to do this. As arguments it requires a target object for the message, some user specific data, and a functor that maps to the specific function in the message interface you wish to target.

```
ExecuteEvents.Execute<ICustomMessageTarget>(target, null, (x,y)=>x.Message1());
```

This code will execute the function Message1 on any components on the GameObject target that implement the ICustomMessageTarget interface. The scripting documentation for the ExecuteEvents class covers other forms of the Execute functions, such as Executing in children or in parents.

Input Modules

Leave feedback

An Input Module is where the main logic of an **event system** can be configured and customised. Out of the box there are two provided Input Modules, one designed for Standalone, and one designed for Touch input. Each module receives and dispatches events as you would expect on the given configuration.

Input modules are where the 'business logic' of the Event System take place. When the Event System is enabled it looks at what Input Modules are attached and passes update handling to the specific module.

Input modules are designed to be extended or modified based on the input systems that you wish to support. Their purpose is to map hardware specific input (such as touch, joystick, mouse, motion controller) into events that are sent via the messaging system.

The built in Input Modules are designed to support common game configurations such as touch input, controller input, keyboard input, and mouse input. They send a variety of events to controls in the application, if you implement the specific interfaces on your MonoBehaviours. All of the **UI** components implement the interfaces that make sense for the given **component**.

Supported Events

Leave feedback

The **Event System** supports a number of events, and they can be customised further in user custom user written Input Modules.

The events that are supported by the Standalone Input Module and Touch Input Module are provided by interface and can be implemented on a MonoBehaviour by implementing the interface. If you have a valid Event System configured the events will be called at the correct time.

IPointerEnterHandler - OnPointerEnter - Called when a pointer enters the object
IPointerExitHandler - OnPointerExit - Called when a pointer exits the object
IPointerDownHandler - OnPointerDown - Called when a pointer is pressed on the object
IPointerUpHandler - OnPointerUp - Called when a pointer is released (called on the GameObject that the pointer is clicking)
IPointerClickHandler - OnPointerClick - Called when a pointer is pressed and released on the same object
IInitializePotentialDragHandler - OnInitializePotentialDrag - Called when a drag target is found, can be used to initialise values
IBeginDragHandler - OnBeginDrag - Called on the drag object when dragging is about to begin
IDragHandler - OnDrag - Called on the drag object when a drag is happening
IEndDragHandler - OnEndDrag - Called on the drag object when a drag finishes
IDropHandler - OnDrop - Called on the object where a drag finishes
IScrollHandler - OnScroll - Called when a mouse wheel scrolls
IUpdateSelectedHandler - OnUpdateSelected - Called on the selected object each tick
ISelectHandler - OnSelect - Called when the object becomes the selected object
IDeselectHandler - OnDeselect - Called on the selected object becomes deselected
IMoveHandler - OnMove - Called when a move event occurs (left, right, up, down, ect)
ISubmitHandler - OnSubmit - Called when the submit button is pressed
ICancelHandler - OnCancel - Called when the cancel button is pressed

Raycasters

Leave feedback

The **Event System** needs a method for detecting where current input events need to be sent to, and this is provided by the Raycasters. Given a screen space position they will collect all potential targets, figure out if they are under the given position, and then return the object that is closest to the screen. There are a few types of Raycasters that are provided:

Graphic Raycaster - Used for **UI** elements, lives on a Canvas and searches within the canvas
Physics 2D Raycaster - Used for 2D physics elements
Physics Raycaster - Used for 3D physics elements

When a Raycaster is present and enabled in the **scene** it will be used by the Event System whenever a query is issued from an Input Module.

If multiple Raycasters are used then they will all have casting happen against them and the results will be sorted based on distance to the elements.

Event System Manager

Leave feedback

This subsystem is responsible for controlling all the other elements that make up eventing. It coordinates which Input Module is currently active, which **GameObject** is currently considered 'selected', and a host of other high level **Event System** concepts.

Each 'Update' the Event System receives the call, looks through its Input Modules and figures out which is the Input Module that should be used for this tick. It then delegates the processing to the modules.

Properties

Property:	*Function:*
First Selected	The GameObject that was selected first.
Send Navigation Events	Should the EventSystem allow navigation events (move / submit / cancel).
Drag Threshold	The soft area for dragging in **pixels**.

Beneath the Properies table is the "Add Default Input Modules" button.

Graphic Raycaster

Leave feedback

The Graphic Raycaster is used to raycast against a Canvas. The Raycaster looks at all Graphics on the canvas and determines if any of them have been hit.

The Graphic Raycaster can be configured to ignore backfacing Graphics as well as be blocked by 2D or **3D objects** that exist in front of it. A manual priority can also be applied if you want processing of this element to be forced to the front or back of the Raycasting.

Properties

Property:	*Function:*
Ignore Reversed Graphics	Should graphics facing away from the raycaster be considered?
Blocked Objects	Type of objects that will block graphic raycasts.
Blocking Mask	Type of objects that will block graphic raycasts.

Physics Raycaster

Leave feedback

The Raycaster raycasts against **3D objects** in the **scene**. This allows messages to be sent to 3D physics objects that implement event interfaces.

Properties

Property:	*Function:*
Depth	Get the depth of the configured **camera**.
Event Camera	Get the camera that is used for this module.
Event Mask	Logical and of Camera mask and eventMask.
Final Event Mask	Logical and of Camera mask and eventMask.

Physics 2D Raycaster

Leave feedback

The 2D Raycaster raycasts against **2D objects** in the **scene**. This allows messages to be sent to 2D physics objects that implement event interfaces. The **Camera GameObject** needs to be used and will be added to the **GameObject** if the Physics 3D Raycaster is not added to the **Camera GameObject**.

For more Raycaster information see Raycasters.

Properties

Property:	*Function:*
Event Camera	The camera that will generate rays for this raycaster.
Priority	Priority of the caster relative to other casters.
Sort Order Priority	Priority of the raycaster based upon sort order.
Render Order Priority	Priority of the raycaster based upon render order.

Standalone Input Module

Leave feedback

The module is designed to work as you would expect a controller / mouse input to work. Events for button presses, dragging, and similar are sent in response to input.

The module sends pointer events to **components** as a mouse / input device is moved around, and uses the Graphics Raycaster and Physics Raycaster to calculate which element is currently pointed at by a given pointer device. You can configure these raycasters to detect or ignore parts of your **Scene**, to suit your requirements.

The module sends move events and submit / cancel events in response to Input tracked via the **Input manager**. This works for both keyboard and controller input. The tracked axis and keys can be configured in the module's **inspector**.

Properties

Property:	*Function:*
Horizontal Axis	Type the desired manager name for the horizontal axis button.
Vertical Axis	Type the desired manager name for the vertical axis.
Submit Button	Type the desired manager name for the Submit button.
Cancel Button	Type the desired manager name for the Cancel button.
Input Actions Per Second	Number of keyboard/controller inputs allowed per second.
Repeat Delay	Delay in seconds before the input actions per second repeat rate takes effect.
Force Module Active	Tick this checkbox to force this **Standalone Input Module** to be active.

Details

The module uses:

Vertical / Horizontal axis for keyboard and controller navigation
Submit / Cancel button for sending submit and cancel events
Has a timeout between events to only allow a maximum number of events a second.
The flow for the module is as follows

Send a Move event to the selected object if a valid axis from the input manager is entered
Send a submit or cancel event to the selected object if a submit or cancel button is pressed
Process Mouse input
If it is a new press
Send PointerEnter event (sent to every object up the hierarchy that can handle it)
Send PointerPress event
Cache the drag handler (first element in the hierarchy that can handle it)
Send BeginDrag event to the drag handler
Set the 'Pressed' object as Selected in the event system
If this is a continuing press
Process movment
Send DragEvent to the cached drag handler
Handle PointerEnter and PointerExit events if touch moves between objects
If this is a release
Send PointerUp event to the object that received the PointerPress
If the current hover object is the same as the PointerPress object send a PointerClick event
Send a Drop event if there was a drag handler cached
Send a EndDrag event to the cached drag handler
Process scroll wheel events

Touch Input Module

Leave feedback

Note: TouchInputModule is obsolete. Touch input is now handled in StandaloneInputModule.

This module is designed to work with touch devices. It sends pointer events for touching and dragging in response to user input. The module supports multitouch.

The module uses the **scene** configured Raycasters to calculate what element is currently being touched over. A raycast is issued for each current touch.

Properties

Property:	*Function:*
Force Module Active	Forces this module to be active.

Details

The flow for the module is as follows:

For each touch event
If it is a new press
Send PointerEnter event (sent to every object up the hierarchy that can handle it)
Send PointerPress event
Cache the drag handler (first element in the hierarchy that can handle it)
Send BeginDrag event to the drag handler
Set the 'Pressed' object as Selected in the event system
If this is a continuing press
Process movment
Send DragEvent to the cached drag handler
Handle PointerEnter and PointerExit events if touch moves between objects
If this is a release
Send PointerUp event to the object that received the PointerPress
If the current hover object is the same as the PointerPress object send a PointerClick event
Send a Drop event if there was a drag handler cached
Send a EndDrag event to the cached drag handler

Event Trigger

Leave feedback

The Event Trigger receives events from the **Event System** and calls registered functions for each event.

The Event Trigger can be used to specify functions you wish to be called for each Event System event. You can assign multiple functions to a single event and whenever the Event Trigger receives that event it will call those functions.

Note that attaching an Event Trigger **component** to a **GameObject** will make that object intercept all events, and no event bubbling will occur from this object!

Events

Each of the Supported Events can optionally be included in the Event Trigger by clicking the Add New Event Type button.

C# Job System

Leave feedback

The Unity C# Job System lets you write simple and safe multithreaded code that interacts with the Unity Engine for enhanced game performance.

You can use the C# Job System with the Unity Entity Component System (ECS), which is an architecture that makes it simple to create efficient machine code for all platforms.

C# Job System Overview
What is multithreading?
What is a job system?
The safety system in the C# Job System
NativeContainer
Creating jobs
Scheduling jobs
JobHandle and dependencies
ParallelFor jobs
ParallelForTransform jobs
C# Job System tips and troubleshooting

2018–03–27 Page published with editorial review

C# Job System exposed in 2018.1

C# Job System Overview

Leave feedback

How the C# Job System works

The Unity C# Job System allows users to write multithreaded code that interacts well with the rest of Unity and makes it easier to write correct code.

Writing multithreaded code can provide high-performance benefits. These include significant gains in frame rate and improved battery life for mobile devices.

An essential aspect of the C# Job System is that it integrates with what Unity uses internally (Unity's native job system). User-written code and Unity share worker threads. This cooperation avoids creating more threads than CPU cores, which would cause contention for CPU resources.

For more information, watch the talk Unity at GDC - Job System & Entity Component System.

2018–06–15 Page published with editorial review

C# Job System exposed in 2018.1

What is multithreading?

Leave feedback

In a single-threaded computing system, one instruction goes in at a time, and one result comes out at a time. The time to load and complete programs depends on the amount of work you need the CPU to do.

Multithreading is a type of programming that takes advantage of a CPU's capability to process many threads at the same time across multiple cores. Instead of tasks or instructions executing one after another, they run simultaneously.

One thread runs at the start of a program by default. This is the "main thread". The main thread creates new threads to handle tasks. These new threads run in parallel to one another, and usually synchronize their results with the main thread once completed.

This approach to multithreading works well if you have a few tasks that run for a long time. However, game development code usually contains many small instructions to execute at once. If you create a thread for each one, you can end up with many threads, each with a short lifetime. That can push the limits of the processing capacity of your CPU and operating system.

It is possible to mitigate the issue of thread lifetime by having a pool of threads. However, even if you use a thread pool, you are likely to have a large number of threads active at the same time. Having more threads than CPU cores leads to the threads contending with each other for CPU resources, which causes frequent context switching as a result. Context switching is the process of saving the state of a thread part way through execution, then working on another thread, and then reconstructing the first thread, later on, to continue processing it. Context switching is resource-intensive, so you should avoid the need for it wherever possible.

2018–06–15 Page published with editorial review

C# Job System exposed in 2018.1

What is a job system?

Leave feedback

A job system manages multithreaded code by creating jobs instead of threads.

A job system manages a group of worker threads across multiple cores. It usually has one worker thread per logical CPU core, to avoid context switching (although it may reserve some cores for the operating system or other dedicated applications).

A job system puts jobs into a job queue to execute. Worker threads in a job system take items from the job queue and execute them. A job system manages dependencies and ensures that jobs execute in the appropriate order.

What is a job?

A job is a small unit of work that does one specific task. A job receives parameters and operates on data, similar to how a method call behaves. Jobs can be self-contained, or they can depend on other jobs to complete before they can run.

What are job dependencies?

In complex systems, like those required for game development, it is unlikely that each job is self-contained. One job is usually preparing the data for the next job. Jobs are aware of and support dependencies to make this work. If `jobA` has a dependency on `jobB`, the job system ensures that `jobA` does not start executing until `jobB` is complete.

2018–06–15 Page published with editorial review

C# Job System exposed in 2018.1

The safety system in the C# Job System

Leave feedback

Race conditions

When writing multithreaded code, there is always a risk for race conditions. A race condition occurs when the output of one operation depends on the timing of another process outside of its control.

A race condition is not always a bug, but it is a source of indeterministic behavior. When a race condition does cause a bug, it can be hard to find the source of the problem because it depends on timing, so you can only recreate the issue on rare occasions. Debugging it can cause the problem to disappear, because breakpoints and logging can change the timing of individual threads. Race conditions produce the most significant challenge in writing multithreaded code.

Safety system

To make it easier to write multithreaded code, the Unity C# Job System detects all potential race conditions and protects you from the bugs they can cause.

For example: if the C# Job System sends a reference to data from your code in the main thread to a job, it cannot verify whether the main thread is reading the data at the same time the job is writing to it. This scenario creates a race condition.

The C# Job System solves this by sending each job a copy of the data it needs to operate on, rather than a reference to the data in the main thread. This copy isolates the data, which eliminates the race condition.

The way the C# Job System copies data means that a job can only access blittable data types. These types do not need conversion when passed between managed and native code.

The C# Job System can copy blittable types with memcpy and transfer the data between the managed and native parts of Unity. It uses `memcpy` to put data into native memory when scheduling jobs and gives the managed side access to that copy when executing jobs. For more information, see Scheduling jobs.

2018–06–15 Page published with editorial review

C# Job System exposed in 2018.1

NativeContainer

Leave feedback

The drawback to the safety system's process of copying data is that it also isolates the results of a job within each copy. To overcome this limitation you need to store the results in a type of shared memory called NativeContainer.

What is a NativeContainer?

A `NativeContainer` is a managed value type that provides a relatively safe C# wrapper for native memory. It contains a pointer to an unmanaged allocation. When used with the Unity C# Job System, a `NativeContainer` allows a job to access data shared with the main thread rather than working with a copy.

What types of NativeContainer are available?

Unity ships with a `NativeContainer` called NativeArray. You can also manipulate a `NativeArray` with NativeSlice to get a subset of the `NativeArray` from a particular position to a certain length.

Note: The Entity Component System (ECS) package extends the `Unity.Collections` namespace to include other types of `NativeContainer`:

`NativeList` - a resizable `NativeArray`.
`NativeHashMap` - key and value pairs.
`NativeMultiHashMap` - multiple values per key.
`NativeQueue` - a first in, first out (FIFO) queue.

NativeContainer and the safety system

The safety system is built into all `NativeContainer` types. It tracks what is reading and writing to any `NativeContainer`.

Note: All safety checks on `NativeContainer` types (such as out of bounds checks, deallocation checks, and race condition checks) are only available in the Unity **Editor** and **Play Mode**.

Part of this safety system is the DisposeSentinel and AtomicSafetyHandle. The `DisposeSentinel` detects memory leaks and gives you an error if you have not correctly freed your memory. Triggering the memory leak error happens long after the leak occurred.

Use the `AtomicSafetyHandle` to transfer ownership of a `NativeContainer` in code. For example, if two scheduled jobs are writing to the same `NativeArray`, the safety system throws an exception with a clear error message that explains why and how to solve the problem. The safety system throws this exception when you schedule the offending job.

In this case, you can schedule a job with a dependency. The first job can write to the `NativeContainer`, and once it has finished executing, the next job can then safely read and write to that same `NativeContainer`. The read and write restrictions also apply when accessing data from the main thread. The safety system does allow multiple jobs to read from the same data in parallel.

By default, when a job has access to a `NativeContainer`, it has both read and write access. This configuration can slow performance. The C# Job System does not allow you to schedule a job that has write access to a `NativeContainer` at the same time as another job that is writing to it.

If a job does not need to write to a `NativeContainer`, mark the `NativeContainer` with the `[ReadOnly]` attribute, like so:

```
[ReadOnly]
public NativeArray<int> input;
```

In the above example, you can execute the job at the same time as other jobs that also have read-only access to the first `NativeArray`.

Note: There is no protection against accessing static data from within a job. Accessing static data circumvents all safety systems and can crash Unity. For more information, see C# Job System tips and troubleshooting.

NativeContainer Allocator

When creating a `NativeContainer`, you must specify the type of memory allocation you need. The allocation type depends on the length of time the job runs. This way you can tailor the allocation to get the best performance possible in each situation.

There are three Allocator types for `NativeContainer` memory allocation and release. You need to specify the appropriate one when instantiating your `NativeContainer`.

Allocator.Temp has the fastest allocation. It is for allocations with a lifespan of one frame or fewer. You should not pass `NativeContainer` allocations using `Temp` to jobs. You also need to call the `Dispose` method before you return from the method call (such as MonoBehaviour.Update, or any other callback from native to managed code).
Allocator.TempJob is a slower allocation than `Temp` but is faster than `Persistent`. It is for allocations within a lifespan of four frames and is thread-safe. If you don't `Dispose` of it within four frames, the console prints a warning, generated from the native code. Most small jobs use this `NativeContainer` allocation type.
Allocator.Persistent is the slowest allocation but can last as long as you need it to, and if necessary, throughout the application's lifetime. It is a wrapper for a direct call to malloc. Longer jobs can use this `NativeContainer` allocation type. You should not use `Persistent` where performance is essential.
For example:

```
NativeArray<float> result = new NativeArray<float>(1, Allocator.TempJob);
```

Note: The number 1 in the example above indicates the size of the `NativeArray`. In this case, it has only one array element (as it only stores one piece of data in `result`).

2018–06–15 Page published with editorial review

C# Job System exposed in 2018.1

Creating jobs

Leave feedback

To create a job in Unity you need to implement the IJob interface. `IJob` allows you to schedule a single job that runs in parallel to any other jobs that are running.

Note: A "job" is a collective term in Unity for any struct that implements the `IJob` interface.

To create a job, you need to:

Create a struct that implements `IJob`.
Add the member variables that the job uses (either blittable types or NativeContainer types).
Create a method in your struct called Execute with the implementation of the job inside it.
When executing the job, the `Execute` method runs once on a single core.

Note: When designing your job, remember that they operate on copies of data, except in the case of `NativeContainer`. So, the only way to access data from a job in the main thread is by writing to a `NativeContainer`.

An example of a simple job definition

```
// Job adding two floating point values together
public struct MyJob : IJob
{
    public float a;
    public float b;
    public NativeArray<float> result;

    public void Execute()
    {
        result[0] = a + b;
    }
}
```

2018–06–15 Page published with editorial review

C# Job System exposed in 2018.1

Scheduling jobs

Leave feedback

To schedule a job in the main thread, you must:

Instantiate the job.
Populate the job's data.
Call the Schedule method.

Calling `Schedule` puts the job into the job queue for execution at the appropriate time. Once scheduled, you cannot interrupt a job.

Note: You can only call `Schedule` from the main thread.

An example of scheduling a job

```
// Create a native array of a single float to store the result. This example wai
NativeArray<float> result = new NativeArray<float>(1, Allocator.TempJob);

// Set up the job data
MyJob jobData = new MyJob();
jobData.a = 10;
jobData.b = 10;
jobData.result = result;

// Schedule the job
JobHandle handle = jobData.Schedule();

// Wait for the job to complete
handle.Complete();

// All copies of the NativeArray point to the same memory, you can access the re
float aPlusB = result[0];

// Free the memory allocated by the result array
result.Dispose();
```

2018–06–15 Page published with editorial review

C# Job System exposed in 2018.1

JobHandle and dependencies

Leave feedback

When you call the Schedule method of a job it returns a JobHandle. You can use a `JobHandle` in your code as a dependency for other jobs. If a job depends on the results of another job, you can pass the first job's `JobHandle` as a parameter to the second job's `Schedule` method, like so:

```
JobHandle firstJobHandle = firstJob.Schedule();
secondJob.Schedule(firstJobHandle);
```

Combining dependencies

If a job has many dependencies, you can use the method JobHandle.CombineDependencies to merge them. `CombineDependencies` allows you to pass them onto the `Schedule` method.

```
NativeArray<JobHandle> handles = new NativeArray<JobHandle>(numJobs, Allocator.T

// Populate `handles` with `JobHandles` from multiple scheduled jobs...

JobHandle jh = JobHandle.CombineDependencies(handles);
```

Waiting for jobs in the main thread

Use `JobHandle` to force your code to wait in the main thread for your job to finish executing. To do this, call the method Complete on the `JobHandle`. At this point, you know the main thread can safely access the NativeContainer that the job was using.

Note: Jobs do not start executing when you schedule them. If you are waiting for the job in the main thread, and you need access to the NativeContainer data that the job is using, you can call the method `JobHandle.Complete`. This method flushes the jobs from the memory cache and starts the process of execution. Calling `Complete` on a `JobHandle` returns ownership of that job's `NativeContainer` types to the main thread. You need to call `Complete` on a `JobHandle` to safely access those `NativeContainer` types from the main thread again. It is also possible to return ownership to the main thread by calling `Complete` on a `JobHandle` that is from a job dependency. For example, you can call `Complete` on `jobA`, or you can call `Complete` on `jobB` which depends on `jobA`. Both result in the `NativeContainer` types that are used by `jobA` being safe to access on the main thread after the call to `Complete`.

Otherwise, if you don't need access to the data, you need to explicity flush the batch. To do this, call the static method JobHandle.ScheduleBatchedJobs. Note that calling this method can negatively impact performance.

An example of multiple jobs and dependencies

Job code:

```
// Job adding two floating point values together
public struct MyJob : IJob
{
    public float a;
    public float b;
    public NativeArray<float> result;

    public void Execute()
    {
        result[0] = a + b;
    }
}

// Job adding one to a value
public struct AddOneJob : IJob
{
    public NativeArray<float> result;

    public void Execute()
    {
        result[0] = result[0] + 1;
    }
}
```

Main thread code:

```
// Create a native array of a single float to store the result in. This example
NativeArray<float> result = new NativeArray<float>(1, Allocator.TempJob);

// Setup the data for job #1
MyJob jobData = new MyJob();
jobData.a = 10;
jobData.b = 10;
jobData.result = result;

// Schedule job #1
JobHandle firstHandle = jobData.Schedule();

// Setup the data for job #2
AddOneJob incJobData = new AddOneJob();
```

```
incJobData.result = result;

// Schedule job #2
JobHandle secondHandle = incJobData.Schedule(firstHandle);

// Wait for job #2 to complete
secondHandle.Complete();

// All copies of the NativeArray point to the same memory, you can access the re
float aPlusB = result[0];

// Free the memory allocated by the result array
result.Dispose();
```

2018–06–15 Page published with editorial review

C# Job System exposed in 2018.1

ParallelFor jobs

Leave feedback

When scheduling jobs, there can only be one job doing one task. In a game, it is common to want to perform the same operation on a large number of objects. There is a separate job type called IJobParallelFor to handle this.

Note: A "ParallelFor" job is a collective term in Unity for any struct that implements the `IJobParallelFor` interface.

A ParallelFor job uses a NativeArray of data to act on as its data source. ParallelFor jobs run across multiple cores. There is one job per core, each handling a subset of the workload. `IJobParallelFor` behaves like `IJob`, but instead of a single Execute method, it invokes the `Execute` method once per item in the data source. There is an integer parameter in the `Execute` method. This index is to access and operate on a single element of the data source within the job implementation.

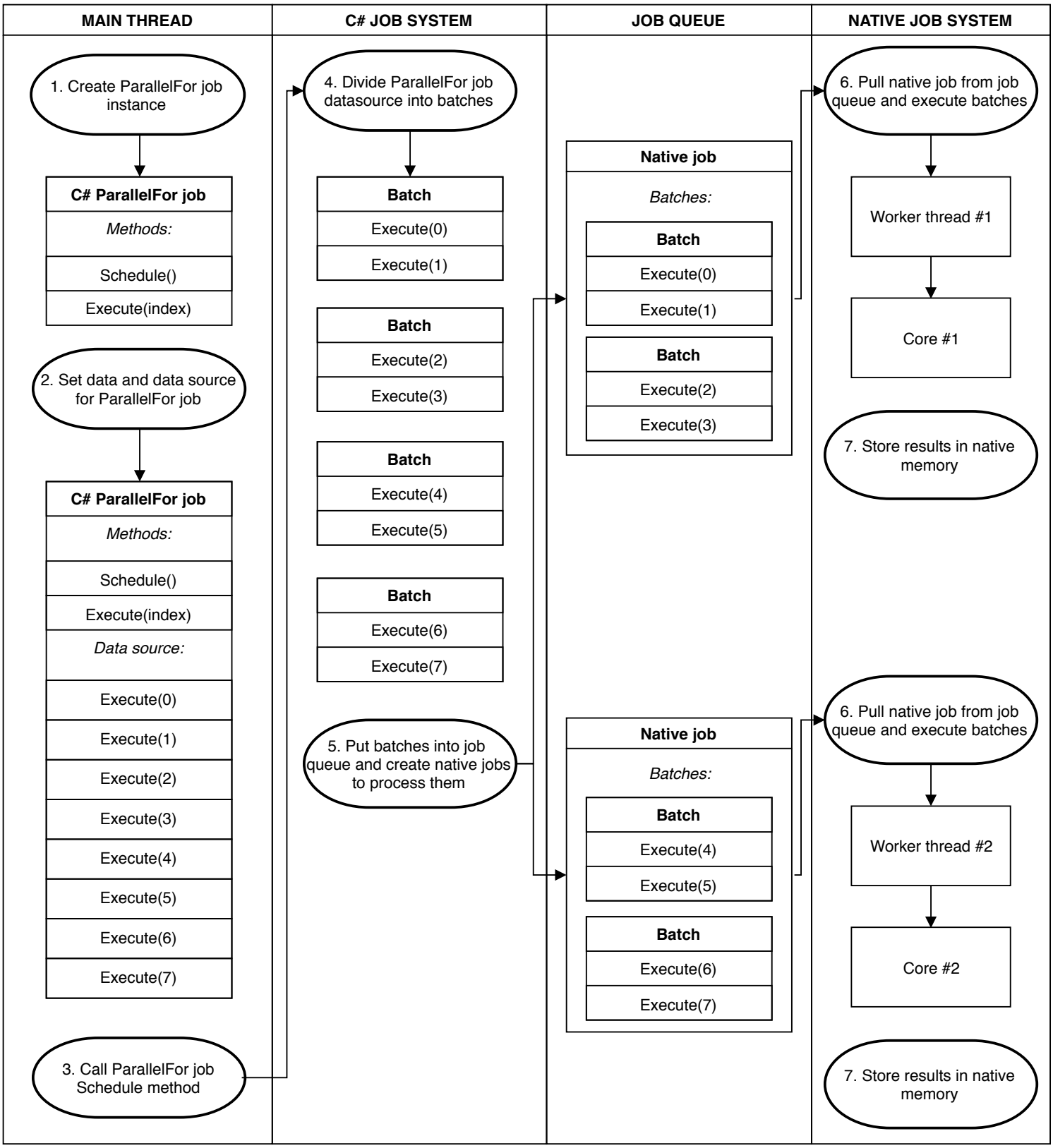
An example of a ParallelFor job definition:

```
struct IncrementByDeltaTimeJob: IJobParallelFor
{
    public NativeArray<float> values;
    public float deltaTime;

    public void Execute (int index)
    {
        float temp = values[index];
        temp += deltaTime;
        values[index] = temp;
    }
}
```

Scheduling ParallelFor jobs

When scheduling ParallelFor jobs, you must specify the length of the `NativeArray` data source that you are splitting. The Unity C# Job System cannot know which `NativeArray` you want to use as the data source if there are several in the struct. The length also tells the C# Job System how many `Execute` methods to expect.

Behind the **scenes**, the scheduling of ParallelFor jobs is more complicated. When scheduling ParallelFor jobs, the C# Job System divides the work into batches to distribute between cores. Each batch contains a subset of `Execute` methods. The C# Job System then schedules up to one job in Unity's native job system per CPU core and passes that native job some batches to complete.



A ParallelFor job dividing batches across cores

When a native job completes its batches before others, it steals remaining batches from the other native jobs. It only steals half of a native job's remaining batches at a time, to ensure cache locality.

To optimize the process, you need to specify a batch count. The batch count controls how many jobs you get, and how fine-grained the redistribution of work between threads is. Having a low batch count, such as 1, gives you a more even distribution of work between threads. It does come with some overhead, so sometimes it is better to

increase the batch count. Starting at 1 and increasing the batch count until there are negligible performance gains is a valid strategy.

An example of scheduling a ParallelFor job

Job code:

```
// Job adding two floating point values together
public struct MyParallelJob : IJobParallelFor
{
    [ReadOnly]
    public NativeArray<float> a;
    [ReadOnly]
    public NativeArray<float> b;
    public NativeArray<float> result;

    public void Execute(int i)
    {
        result[i] = a[i] + b[i];
    }
}
```

Main thread code:

```
NativeArray<float> a = new NativeArray<float>(2, Allocator.TempJob);

NativeArray<float> b = new NativeArray<float>(2, Allocator.TempJob);

NativeArray<float> result = new NativeArray<float>(2, Allocator.TempJob);

a[0] = 1.1;
b[0] = 2.2;
a[1] = 3.3;
b[1] = 4.4;

MyParallelJob jobData = new MyParallelJob();
jobData.a = a;
jobData.b = b;
jobData.result = result;

// Schedule the job with one Execute per index in the results array and only 1 i
JobHandle handle = jobData.Schedule(result.Length, 1);
```

```
// Wait for the job to complete
handle.Complete();

// Free the memory allocated by the arrays
a.Dispose();
b.Dispose();
result.Dispose();
```

2018–06–15 Page published with editorial review

C# Job System exposed in 2018.1

ParallelForTransform jobs

Leave feedback

A ParallelForTransform job is another type of ParallelFor job; designed specifically for operating on Transforms.

Note: A ParallelForTransform job is a collective term in Unity for any job that implements the IJobParallelForTransform interface.

2018–06–15 Page published with editorial review

C# Job System exposed in 2018.1

C# Job System tips and troubleshooting

Leave feedback

When using the Unity C# Job System, make sure you adhere to the following:

Do not access static data from a job

Accessing static data from a job circumvents all safety systems. If you access the wrong data, you might crash Unity, often in unexpected ways. For example, accessing MonoBehaviour can cause crashes on domain reloads.

Note: Because of this risk, future versions of Unity will prevent global variable access from jobs using static analysis. If you do access static data inside a job, you should expect your code to break in future versions of Unity.

Flush scheduled batches

When you want your jobs to start executing, then you can flush the scheduled batch with JobHandle.ScheduleBatchedJobs. Note that calling this method can negatively impact performance. Not flushing the batch delays the scheduling until the main thread waits for the result. In all other cases use JobHandle.Complete to start the execution process.

Note: In the Entity Component System (ECS) the batch is implicitly flushed for you, so calling `JobHandle.ScheduleBatchedJobs` is not necessary.

Don't try to update NativeContainer contents

Due to the lack of ref returns, it is not possible to directly change the content of a NativeContainer. For example, `nativeArray[0]++;` is the same as writing `var temp = nativeArray[0]; temp++;` which does not update the value in `nativeArray`.

Instead, you must copy the data from the index into a local temporary copy, modify that copy, and save it back, like so:

```
MyStruct temp = myNativeArray[i];
temp.memberVariable = 0;
myNativeArray[i] = temp;
```

Call JobHandle.Complete to regain ownership

Tracing data ownership requires dependencies to complete before the main thread can use them again. It is not enough to check JobHandle.IsCompleted. You must call the method `JobHandle.Complete` to regain ownership of the `NativeContainer` types to the main thread. Calling `Complete` also cleans up the state in the safety system. Not doing so introduces a memory leak. This process also applies if you schedule new jobs every frame that has a dependency on the previous frame's job.

Use Schedule and Complete in the main thread

You can only call Schedule and `Complete` from the main thread. If one job depends on another, use `JobHandle` to manage dependencies rather than trying to schedule jobs within jobs.

Use Schedule and Complete at the right time

Call `Schedule` on a job as soon as you have the data it needs, and don't call `Complete` on it until you need the results. It is good practice to schedule a job that you do not need to wait for when it is not competing with any other jobs that are running. For example, if you have a period between the end of one frame and the beginning of the next frame where no jobs are running, and a one frame latency is acceptable, you can schedule the job towards the end of a frame and use its results in the following frame. Alternatively, if your game is saturating that changeover period with other jobs, and there is a big under-utilized period somewhere else in the frame, it is more efficient to schedule your job there instead.

Mark NativeContainer types as read-only

Remember that jobs have read and write access to `NativeContainer` types by default. Use the `[ReadOnly]` attribute when appropriate to improve performance.

Check for data dependencies

In the Unity **Profiler** window, the marker "WaitForJobGroup" on the main thread indicates that Unity is waiting for a job on a worker thread to complete. This marker could mean that you have introduced a data dependency somewhere that you should resolve. Look for `JobHandle.Complete` to track down where you have data dependencies that are forcing the main thread to wait.

Debugging jobs

Jobs have a Run function that you can use in place of `Schedule` to immediately execute the job on the main thread. You can use this for debugging purposes.

Do not allocate managed memory in jobs

Allocating managed memory in jobs is incredibly slow, and the job is not able to make use of the Unity Burst compiler to improve performance. Burst is a new LLVM based backend compiler technology that makes things easier for you. It takes C# jobs and produces highly-optimized machine code taking advantage of the particular capabilities of your platform.

Further Information

Watch the Unity GDC 2018: C# Job System playlist of clips.
For more advanced information on how the C# Job System relates to ECS, see the ECS package documentation on GitHub.

2018–06–15 Page published with editorial review

C# Job System exposed in 2018.1

Multiplayer and Networking

Leave feedback

This section has an overview and detailed reference pages on making your project multiplayer.

Related tutorials: Multiplayer Networking

Multiplayer Overview

Leave feedback

Related tutorials: Multiplayer Networking

There are two kinds of users for the **Networking** feature:

Users making a Multiplayer game with Unity. These users should start with the NetworkManager or the High Level API.
Users building network infrastructure or advanced multiplayer games. These users should start with the NetworkTransport API.

High level scripting API

Unity's networking has a "high-level" scripting API (which we'll refer to as the HLAPI). Using this means you get access to commands which cover most of the common requirements for multiuser games without needing to worry about the "lower level" implementation details. The HLAPI allows you to:

Control the networked state of the game using a "**Network Manager**".
Operate "client hosted" games, where the host is also a player client.
Serialize data using a general-purpose serializer.
Send and receive network messages.
Send networked commands from clients to servers.
Make remote procedure calls (RPCs) from servers to clients.
Send networked events from servers to clients.

Engine and Editor integration

Unity's networking is integrated into the engine and the editor, allowing you to work with **components** and visual aids to build your multiplayer game. It provides:

A NetworkIdentity component for networked objects.
A NetworkBehaviour for networked **scripts**.
Configurable automatic synchronization of object transforms.
Automatic synchronization of script variables.
Support for placing networked objects in Unity **scenes**.
Network components

Internet Services

Unity offers Internet Services to support your game throughout production and release, which includes:

Matchmaking service
Create matches and advertise matches.
List available matches and join matches.
Relay server
Game-play over internet with no dedicated server.
Routing of messages for participants of matches.

NetworkTransport real-time transport layer

We include a Real-Time Transport Layer that offers:

Optimized UDP based protocol.
Multi-channel design to avoid head-of-line blocking issues
Support for a variety of levels of Quality of Service (QoS) per channel.
Flexible network topology that supports peer-to-peer or client-server architectures.

Sample Projects

You can also dig into our multiplayer sample projects to see how these features are used together. The following sample projects can be found within this Unity Forum post:

Multiplayer 2D Tanks example game
Multiplayer Invaders game with Matchmaking
Multiplayer 2D space shooter with Matchmaking
Minimal Multiplayer project

Setting up a multiplayer project

Leave feedback

This page contains an overview of the most basic and common things you need when setting up a multiplayer project. In terms of what you require in your project, these are:

A **Network Manager**

A **user interface** (for players to find and join games)

Networked **Player Prefabs** (for players to control)

Scripts and **GameObjects** which are **multiplayer-aware**

There are variations on this list; for example, in a multiplayer chess game, or a real-time strategy (RTS) game, you don't need a visible GameObject to represent the player. However, you might still want an invisible empty GameObject to represent the player, and attach scripts to it which relate to what the player is able to do.

This introductory page contains a brief description of each of the items listed above. However, each section links to more detailed documentation, which you need to continue reading to fully understand them.

There are also some important concepts that you need to understand and make choices about when building your game. These concepts can broadly be summarised as:

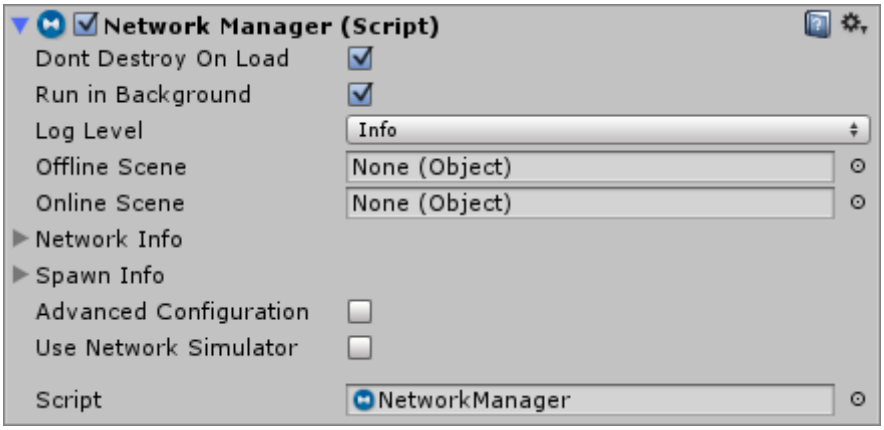
The relationship between a **client**, a **server**, and a **host**

The idea of **authority** over GameObjects and actions

To learn about these concepts, see documentation on Network System Concepts.

The Network Manager

The Network Manager is responsible for managing the **networking** aspects of your multiplayer game. You should have one (and only one) Network Manager active in your **Scene** at a time.



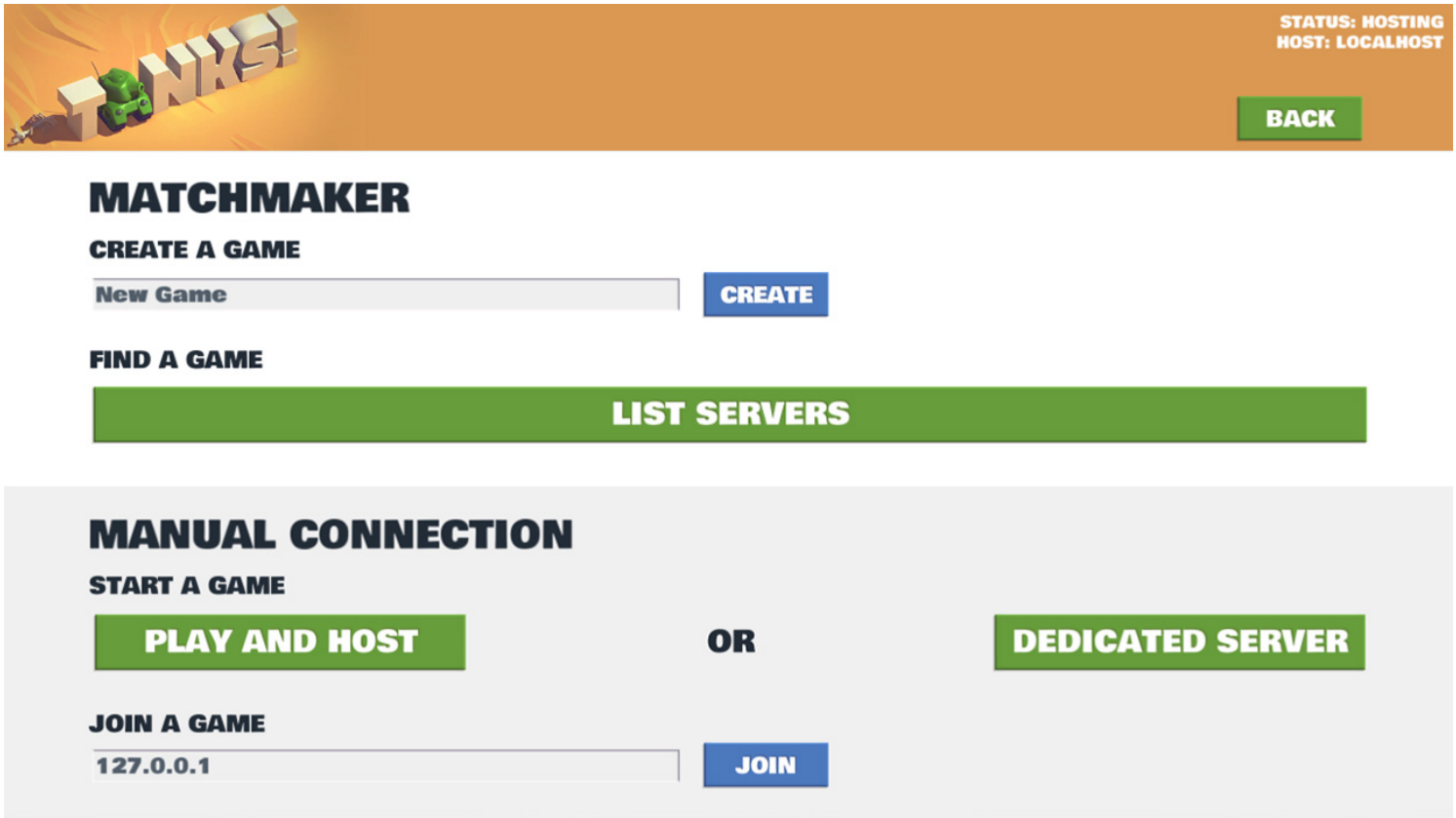
The Network Manager Component

Unity's built-in Network Manager **component** wraps up all of the features for managing your multiplayer game into one single **component**. If you have custom requirements which aren't covered by this **component**, you can write your own network manager in script instead of using this **component**. If you're just starting out with multiplayer games, you should use this **component**.

To learn more, see documentation on the Network Manager.

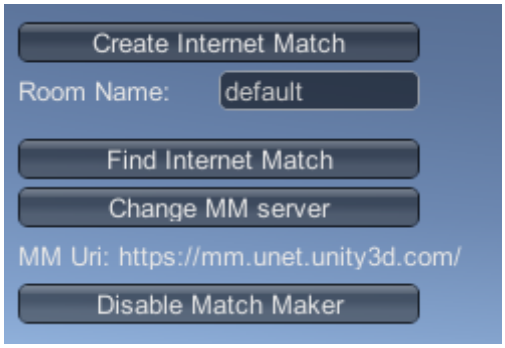
A user interface for players to find and join games

Almost every multiplayer game provides players with a way to discover, create, and join individual game "instances" (also known as "matches"). This part of the game is commonly known as the "lobby", and sometimes has extra features like chat.



A typical multiplayer game lobby, allowing players to find, create and join games, as seen in the TANKS networking demo, available on the Asset Store.

Unity has an extremely basic built-in version of such an interface, called the **NetworkManagerHUD**. It can be extremely useful in the early stages of creating your game, because it allows you to easily create matches and test your game without needing to implement your own **UI**. However, it is very basic in both functionality and visual design, so you should replace this with your own **UI** before you finish your project.



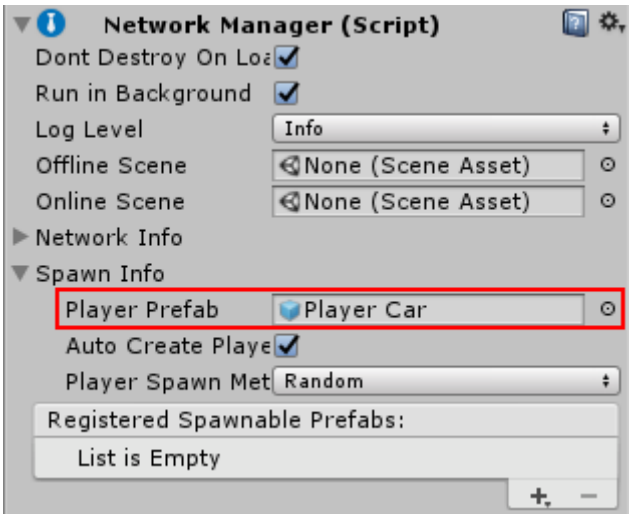
Unity's built-in Network Manager HUD, shown in MatchMaker mode.

To learn more, see documentation on the Network Manager HUD.

Networked player GameObjects

Most multiplayer games feature some kind of object that a player can control, like a character, a car, or something else. Some multiplayer games don't feature a single visible "**player object**" but instead allow a player to control many units or items, like in chess or real-time strategy games. Others don't even feature specific objects at all, like a shared-canvas painting game. In all of these situations, however, you usually need to create a GameObject that *conceptually* represents the player in your game. Make this GameObject a Prefab, and attach all the scripts to it which control what the player can do in your game.

If you are using Unity's Network Manager component (see *The Network Manager*, above), assign the Prefab to the **Player Prefab** field.



The network manager with a "Player Car" prefab assigned to the Player Prefab field.

When the game is running, the Network Manager creates a copy (an "instance") of your player Prefab for each player that connects to the match.

However - and this is where it can get confusing for people new to multiplayer programming - you need to make sure the scripts on your player Prefab instance are "aware" of whether the player controlling the instance is using the **host** computer (the computer that is managing the game) or a **client** computer (a different computer to the one that is managing the game).

This is because both situations will be occurring at the same time.

Multiplayer-aware Scripts

Writing scripts for a multiplayer game is different to writing scripts for a single-player game. This is because when you write a script for a multiplayer game, you need to think about the different contexts that the scripts run in. To learn about the networking concepts discussed here, see documentation on Network System Concepts.

For example, the scripts you place on your player Prefab should allow the "owner" of that player instance to control it, but it should not allow other people to control it.

You need to think about whether the server or the client has authority over what the script does. Sometimes, you want the script to run on both the server and the clients. Other times, you only want the script to run on the server, and you only want the clients to replicate how the GameObjects are moving (for example, in a game in which players pick up collectible GameObjects, the script should only run on the server so that the server can be the authority on the number of GameObjects collected).

Depending on what your script does, you need to decide which parts of your script should be active in which situations.

For player GameObjects, each person usually has active control over their own player instance. This means each client has local authority over its own player, and the server accepts what the client tells it about what the player is doing.

For non-player GameObjects, the server usually has authority over what happens (such as whether an item has been collected), and all clients accept what the server tells them about what has happened to that GameObject.

Using the Network Manager

Leave feedback

The **Network Manager** is a **component** for managing the **networking** aspects of a multiplayer game.

The Network Manager features include:

Game state management

Spawn management

Scene management

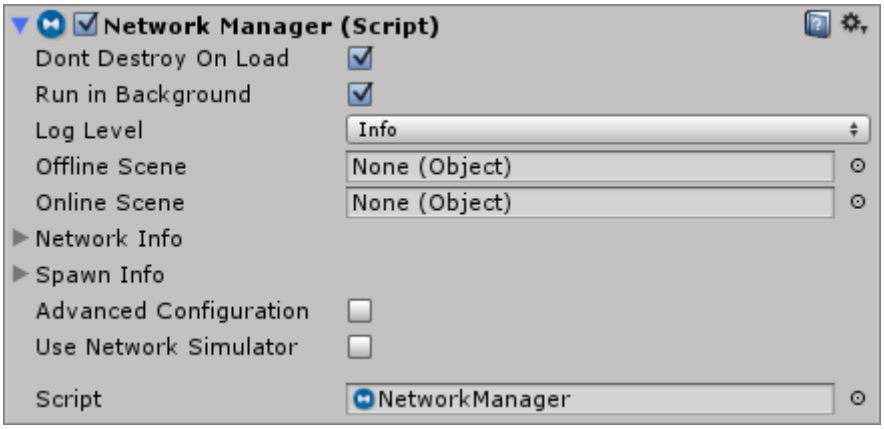
Debugging information

Matchmaking

Customization

Getting Started with the Network Manager

The Network Manager is the core controlling component of a multiplayer game. To get started, create an empty **GameObject** in your starting Scene, and add the NetworkManager component. The newly added Network Manager component looks like this:



The Network Manager as seen in the inspector window

The **Inspector** for the Network Manager in the Editor allows you to configure and control many things related to networking.

Note: You should only ever have one active Network Manager in each Scene. Do not place the Network Manager component on a networked GameObject (one which has a Network Identity component), because Unity disables these when the Scene loads.

If you are already familiar with multiplayer game development, you might find it useful to know that the Network Manager component is implemented entirely using the High-level API (HLAPI), so everything it does is also available to you through scripting. For advanced users, if you find that you need to expand on the Network Manager component's features, you can use scripting to derive your own class from NetworkManager and customize its behaviour by overriding any of the virtual function hooks that it provides. However, the Network Manager component wraps up a lot of useful functionality into a single place, and makes creating, running and debugging multiplayer games as simple as possible.

Game state management

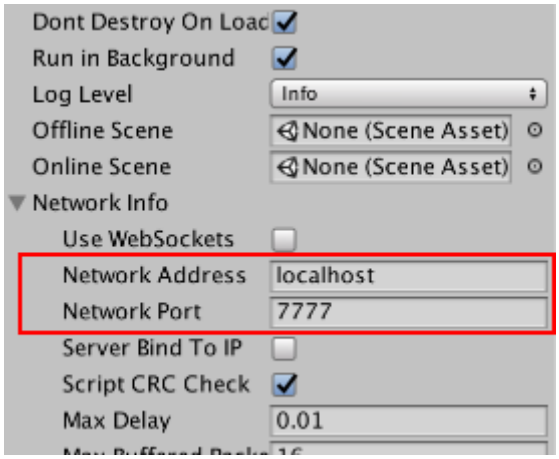
A Networking multiplayer game can run in three modes - as a client, as a dedicated server, or as a “Host” which is both a client and a server at the same time.

If you’re using the Network Manager HUD, it automatically tells the Network Manager which mode to start in, based on which options the player selects. If you’re writing your own **UI** that allows the player to start the game, you’ll need to call these from your own code. These methods are:

NetworkManager.StartClient

NetworkManager.StartServer

NetworkManager.StartHost



The network address and port settings in the Network Manager component

Whichever mode the game starts in (client, server, or host), the **Network Address** and **Network Port** properties are used. In client mode, the game attempts to connect to the address and port specified. In server or host mode, the game listens for incoming connections on the port specified.

During development of your game, it can be useful to put a fixed address and port setting into these properties. However, eventually you might want your players to be able to choose the host they to connect to. When you get to that stage, the Network Discovery component (see Local Discovery) can be used for broadcasting and finding addresses and ports on a local area network (LAN), and the Matchmaker service can be used for players to find internet matches to connect to (see Multiplayer Service).

Spawn management

Use the Network Manager to manage the spawning (networked instantiation) of networked GameObjects from **Prefabs**.

The “Spawn Info” section of the Network Manager component

Most games have a Prefab which represents the player, so the Network Manager has a **Player Prefab** slot. You should assign this slot with your player Prefab. When you have a player Prefab set, a player GameObject is automatically spawned from that Prefab for each user in the game. This applies to the local player on a hosted server, and remote players on remote clients. You must attach a Network Identity component to the Player Prefab.

Once you have assigned a player Prefab, you can start the game as a host and see the player GameObject spawn. Stopping the game destroys the player GameObject. If you build and run another copy of the game and connect it as a client to *localhost*, the Network Manager makes another player GameObject appear. When you stop that client, it destroys that player’s GameObject.

In addition to the player Prefab, you must also register other Prefabs that you want to dynamically spawn during gameplay with the Network Manager.

You can add Prefabs to the list shown in the inspector labelled **Registered Spawnable Prefabs**. You can also can register prefabs via code, with the ClientScene.RegisterPrefab() method.

If you have only one Network Manager, you need to register to it all prefabs which might be spawned in any Scene. If you have a separate Network Manager in each Scene, you only need to register the prefabs relevant for that Scene.

Customizing Player Instantiation

The Network Manager spawns player GameObjects using its implementation of NetworkManager.OnServerAddPlayer(). If you want to customize the way player GameObjects are created, you can override that virtual function. This code shows an example of the default implementation:

```
public virtual void OnServerAddPlayer(NetworkConnection conn, short playerContro
{
    var player = (GameObject)GameObject.Instantiate(playerPrefab, playerSpawnPos
    NetworkServer.AddPlayerForConnection(conn, player, playerControllerId);
}
```

Note: If you are implementing a custom version of OnServerAddPlayer, the method NetworkServer.AddPlayerForConnection() must be called for the newly created player GameObject, so that it is spawned and associated with the client’s connection. AddPlayerForConnection spawns the GameObject, so you do not need to use NetworkServer.Spawn().

Start positions

To control where players are spawned, you can use the Network Start Position component. To use these, attach a Network Start Position component to a GameObject in the Scene, and position the GameObject where you would like one of the players to start. You can add as many start positions to your Scene as you like. The Network

Manager detects all start positions in your Scene, and when it spawns each player instance, it uses the position and orientation of one of them.

The Network Manager has a **Player Spawn Method** property, which allows you to configure how start positions are chosen.

Choose **Random** to spawn players at randomly chosen startPosition options.

Choose **Round Robin** to cycle through startPosition options in a set list.

If the Random or Round Robin modes don't suit your game, you can customize how the start positions are selected by using code. You can access the available Network Start Position components by the list NetworkManager.startPositions, and you can use the helper method GetStartPosition() on the Network Manager that can be used in implementation of OnServerAddPlayer to find a start position.

Scene management

Most games have more than one Scene. At the very least, there is usually a title screen or starting menu Scene in addition to the Scene where the game is actually played. The Network Manager is designed to automatically manage Scene state and Scene transitions in a way that works for a multiplayer game.

There are two slots on the NetworkManager Inspector for scenes: the Offline Scene and the Online Scene. Dragging Scene **assets** into these slots activates networked Scene management.

When a server or host is started, the Online Scene is loaded. This then becomes the current network Scene. Any clients that connect to that server are instructed to also load that Scene. The name of this Scene is stored in the networkSceneName property.

When the network is stopped, by stopping the server or host or by a client disconnecting, the offline Scene is loaded. This allows the game to automatically return to a menu Scene when disconnected from a multiplayer game.

You can also change Scenes while the game is active by calling NetworkManager.ServerChangeScene(). This makes all the currently connected clients change Scene too, and updates networkSceneName so that new clients also load the new Scene.

While networked Scene management is active, any calls to game state management functions such NetworkManager.StartHost() or NetworkManager.StopClient() can cause Scene changes. This applies to the runtime control UI. By setting up Scenes and calling these methods, you can control the flow of your multiplayer game.

Note that Scene changes cause all the GameObjects in the previous Scene to be destroyed.

You should normally make sure the Network Manager persists between Scenes, otherwise the network connection is broken upon a Scene change. To do this, ensure the **Don't Destroy On Load** box is checked in the Inspector. However it is also possible to have a separate Network Manager in each Scene with different settings, which may be helpful if you wish to control incremental Prefab loading, or different Scene transitions.

Customization

There are virtual functions on the NetworkManager class that you can customize by creating your own derived class that inherits from NetworkManager. When implementing these functions, be sure to take care of the functionality that the default implementations provide. For example, in OnServerAddPlayer(), the function NetworkServer.AddPlayer must be called to activate the player GameObject for the connection.

These are all the callbacks that can happen for host/server and clients, in some cases it's important to invoke the base class function to maintain default behaviour. To see the implementation itself you can view it in the networking bitbucket repository.

```
using UnityEngine;

using UnityEngine.Networking;

using UnityEngine.Networking.Match;

public class CustomManager : NetworkManager {

    // Server callbacks

    public override void OnServerConnect(NetworkConnection conn) {

        Debug.Log("A client connected to the server: " + conn);

    }

    public override void OnServerDisconnect(NetworkConnection conn) {

        NetworkServer.DestroyPlayersForConnection(conn);

        if (conn.lastError != NetworkError.Ok) {

            if (LogFilter.logError) { Debug.LogError("ServerDisconnected due to

        }

        Debug.Log("A client disconnected from the server: " + conn);

    }

    public override void OnServerReady(NetworkConnection conn) {

        NetworkServer.SetClientReady(conn);

        Debug.Log("Client is set to the ready state (ready to receive state upda
```

```
    }
    public override void OnServerAddPlayer(NetworkConnection conn, short playerC
        var player = (GameObject)GameObject.Instantiate(playerPrefab, Vector3.ze
        NetworkServer.AddPlayerForConnection(conn, player, playerControllerId);
        Debug.Log("Client has requested to get his player added to the game");
    }
    public override void OnServerRemovePlayer(NetworkConnection conn, PlayerCont
        if (player.gameObject != null)
            NetworkServer.Destroy(player.gameObject);
    }
    public override void OnServerError(NetworkConnection conn, int errorCode) {
        Debug.Log("Server network error occurred: " + (NetworkError)errorCode);
    }
    public override void OnStartHost() {
        Debug.Log("Host has started");
    }
    public override void OnStartServer() {
        Debug.Log("Server has started");
    }
    public override void OnStopServer() {
        Debug.Log("Server has stopped");
    }
    public override void OnStopHost() {
```

```
        Debug.Log("Host has stopped");
    }
    // Client callbacks
    public override void OnClientConnect(NetworkConnection conn)
    {
        base.OnClientConnect(conn);
        Debug.Log("Connected successfully to server, now to set up other stuff f
    }
    public override void OnClientDisconnect(NetworkConnection conn) {
        StopClient();
        if (conn.lastError != NetworkError.Ok)
        {
            if (LogFilter.logError) { Debug.LogError("ClientDisconnected due to
        }
        Debug.Log("Client disconnected from server: " + conn);
    }
    public override void OnClientError(NetworkConnection conn, int errorCode) {
        Debug.Log("Client network error occurred: " + (NetworkError)errorCode);
    }
    public override void OnClientNotReady(NetworkConnection conn) {
        Debug.Log("Server has set client to be not-ready (stop getting state upd
    }
    public override void OnStartClient(NetworkClient client) {
```

```
        Debug.Log("Client has started");

    }

    public override void OnStopClient() {

        Debug.Log("Client has stopped");

    }

    public override void OnClientSceneChanged(NetworkConnection conn) {

        base.OnClientSceneChanged(conn);

        Debug.Log("Server triggered scene change and we've done the same, do any

    }

}
```

The inspector for the NetworkManager provides the ability to change some connection parameters and timeouts. Some parameters have not been exposed here but can be changed through code.

```
using UnityEngine;

using UnityEngine.Networking;

public class CustomManager : NetworkManager {

    // Set custom connection parameters early, so they are not too late to be en

    void Start()

    {

        customConfig = true;

        connectionConfig.MaxCombinedReliableMessageCount = 40;

        connectionConfig.MaxCombinedReliableMessageSize = 800;
```

```
        connectionConfig.MaxSentMessageQueueSize = 2048;

        connectionConfig.IsAcksLong = true;

        globalConfig.ThreadAwakeTimeout = 1;

    }

}
```

Using the Network Manager HUD

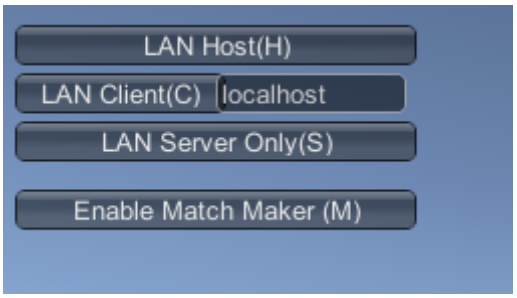
Leave feedback

Note: This documentation assumes that you understand fundamental **networking** concepts such as the relationship between a host, server and client. To learn more about these concepts, see documentation on Network System Concepts.

The Network Manager HUD component, as viewed in the inspector

Property:	*Function:*
Show Runtime GUI	Tick this checkbox to show the **Network Manager** HUD GUI at run time. This allows you to reveal or hide it for quick debugging.
GUI Horizontal Offset	Set the horizontal **pixel** offset of the HUD, measured from the left edge of the screen.
GUI Vertical Offset	Set the vertical pixel offset of the HUD, measured from the top edge of the screen.

The **Network Manager HUD** ("heads-up display") provides the basic functions so that people playing your game can start hosting a networked game, or find and join an existing networked game. Unity displays the Network Manager HUD as a collection of simple **UI** buttons in the Game view.



The Network Manager HUD UI, as viewed in the Game view

The Network Manager HUD is a quick-start tool to help you start building your multiplayer game straight away, without first having to build a user interface for game creation/connection/joining. It allows you to jump straight into your gameplay programming, and means you can build your own version of these controls later in your development schedule.

It is not, however, intended to be included in finished games. The idea is that these controls are useful to get you started, but you should create your own UI later on, to allow your players to find and join games in a way that suits your game. For example, you might want to stylize the design of the screens, buttons and list of available games to match the overall style of your game.

To start using the Network Manager HUD, create an empty GameObject in your Scene (menu: **GameObject** > **Create Empty**) and add the Network Manager HUD component to the new GameObject.

For a description of the properties shown in the **inspector** for the Network Manager HUD, see the Network Manager HUD reference page.

Using the HUD

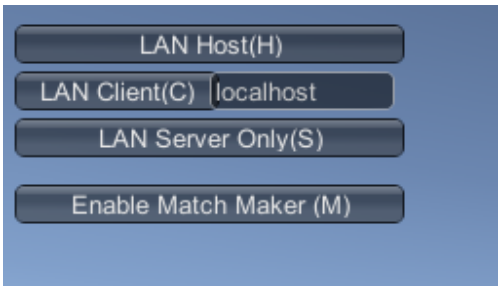
The Network Manager HUD has two basic modes: **LAN** (Local Area Network) mode and **Matchmaker** mode. These modes match the two common types of multiplayer games. **LAN** mode is for creating or joining games hosted on a local area network (that is, multiple computers connected to the same network), and **Matchmaker** mode is for creating, finding and joining games across the internet (multiple computers connected to separate networks).

The Network Manager HUD starts in **LAN** mode, and displays buttons relating to hosting and joining a LAN-based multiplayer game. To switch the HUD to **Matchmaker** mode, click the **Enable Match Maker (M)** button.

Note: Remember that the Network Manager HUD feature is a temporary aid to development. It allows you to get your multiplayer game running quickly, but you should replace it with your own UI controls when you are ready.

The Network Manager HUD in LAN mode

Leave feedback

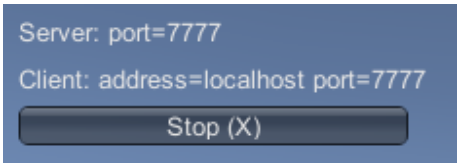


The Network Manager HUD in LAN mode (the default mode) as seen in the Game view.

LAN Host

Click the **LAN Host** button to start a game as a host on the local network. This client is both the host *and* one of the players in the game. It uses the information from the **Network Info** section in the **inspector** to host the game.

When you click this button, the HUD switches to a simple display of network details, and a **Stop (X)** button which allows you to stop hosting the game and return to the main LAN menu.



The Network Manager HUD when hosting a LAN game.

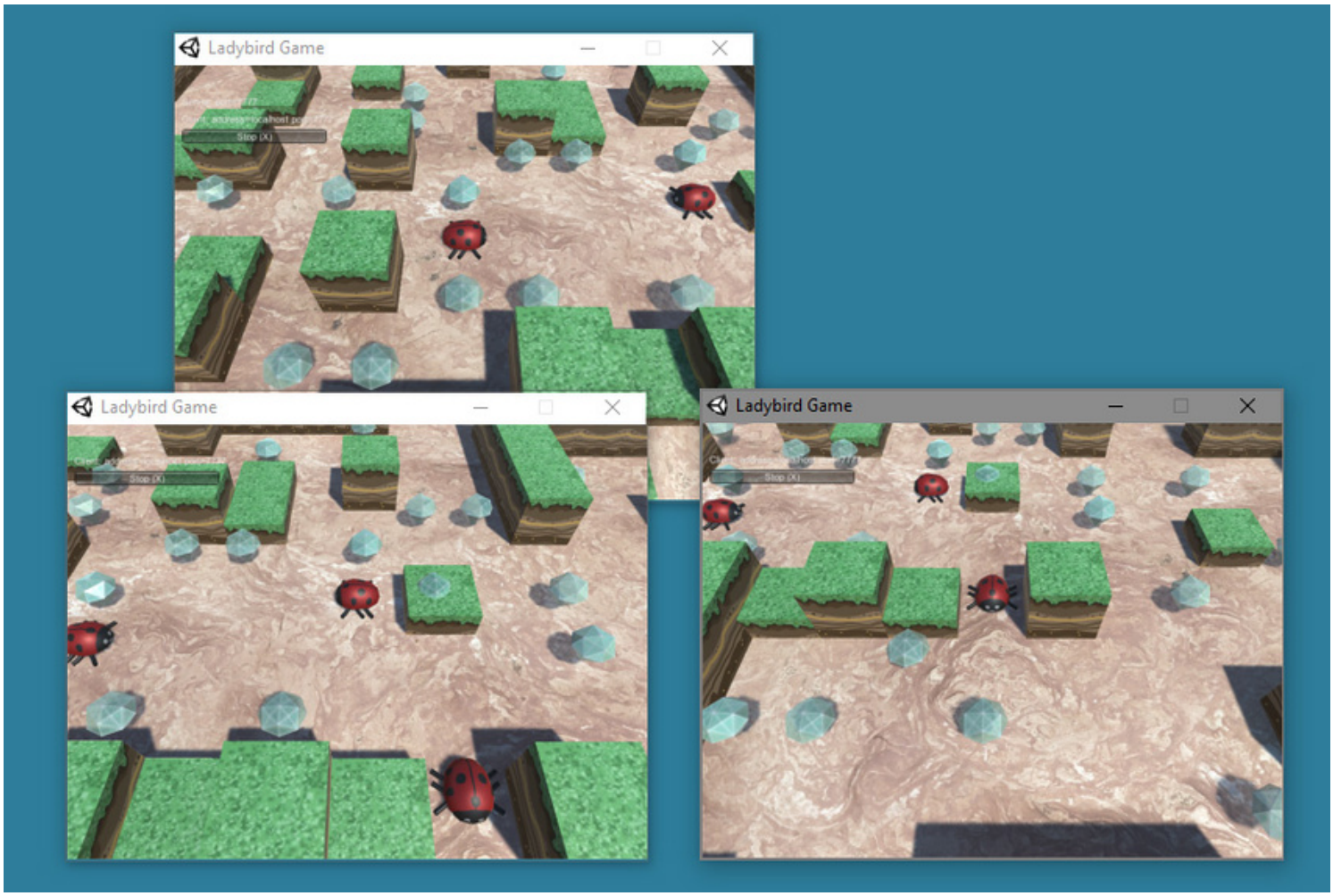
When you have started a game as a host, other players of the game can then connect to the host to join the game.

Click the **Stop (X)** button to disconnect any players that are connected to the host player. Clicking **Stop (X)** also returns the HUD to the LAN menu.

LAN Client

To connect to a host on the local network, use the text field to the right of the **LAN Client** button to specify the address of the host. The default host address is “localhost”, which means the client looks on its own computer for the game host. Click **LAN Client (C)** to attempt to connect to the host address you have specified.

Use the default “localhost” in this field if you are running multiple instances of your game on one computer, to test multiplayer interactivity. To do this, you can create a standalone build of your game, and then launch it multiple times on your computer. This is a common way to quickly test that your networked game interactions are functioning as you expect, without you needing to deploy your game to multiple computers or devices.



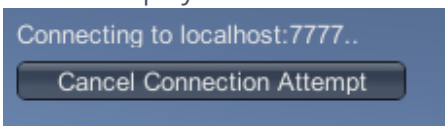
An example of three instances of a networked game running on the same desktop PC. This is useful for quick tests to ensure networked interactions are behaving as you intended. One is running as LAN Host, and two are running as LAN Client.

When you want to test your game on multiple machines within the same network (that is, on a LAN), you need to put the address of the person acting as host into the "localhost" text field.

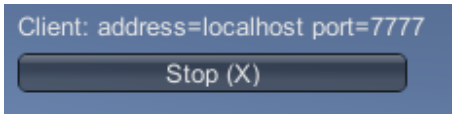
The person acting as the host needs to tell their IP address to everyone running LAN clients, so that you can type this into the box.

Enter the IP address (or leave it as “localhost” if you are testing it on your own machine), then click **LAN Client** to attempt to connect to the host.

When the client is attempting to connect, the HUD displays a **Cancel Connection Attempt** button. Click this if you want to stop trying to connect to the host.



If the connection is successful, the HUD displays the **Stop (X) button**. Click this if you want to stop the game on the client and disconnect from the host:



The HUD after a successful connection

Unity has a built-in Network Discovery system, which allows clients to automatically find hosts on the same local network. However, this is not built into the **Network Manager** HUD, so you need to enter the address manually. You can integrate the Network Discovery system into your game when you replace the **Network Manager** HUD with your own **UI**. See documentation on Network Discovery to learn more.

LAN Server Only

Click **LAN Server Only** to start a game which acts as a server that other clients can connect to, but which does not act as a client to the game itself. This type of game is often called a "dedicated server". A user cannot play the game on this particular instance of your game. All players must connect as clients, and nobody plays on the instance that is running as the server.

A dedicated server on a LAN results in better performance for all connected players, because the server doesn't need to process a local player's gameplay in addition to acting as server.

You might also choose this option if you want to host a game that can be played over the internet (rather than just within a local network), but want to maintain control of the server yourself - for example, to prevent cheating by one of the clients, because only the server has authority over the game. To do this, you would need to run the game in **Server Only** mode on a computer with a public IP address.

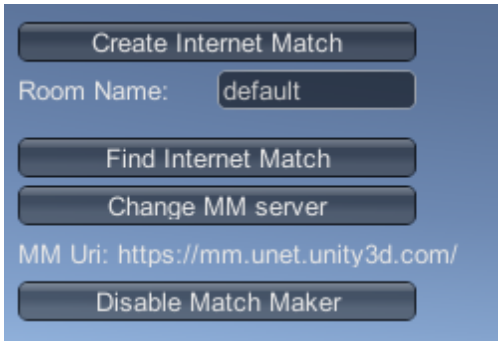
Enable Match Maker

Click **Enable Match Maker (M)** to change the HUD to Matchmaker mode. You need to use Matchmaker mode if you want to create or connect to games hosted on the internet using Unity's Matchmaker multiplayer service. Click **Enable Match Maker (M)** to display the Matchmaker controls in the Network Manager HUD.

Note: Remember that the Network Manager HUD feature is a temporary aid to development. It allows you to get your multiplayer game running quickly, but you should replace it with your own UI controls when you are ready.

The Network Manager HUD in Matchmaker mode

Leave feedback



The Network Manager HUD in Matchmaker mode

Matchmaker mode provides a simple interface that allows players to create, find and join matches hosted on Unity's Multiplayer Service.

A "match" (also sometimes referred to as a game session, or a game instance), is a unique instance of your game hosted by Unity's Multiplayer Service. With Unity's Multiplayer Service, a certain limited number of players can join and play together. If lots of people are playing your game, you may have multiple matches, each with multiple players playing together.

In order to use Matchmaker mode, you must first enable Unity Multiplayer Service for your project. Once you have enabled Unity Multiplayer Service for your project, you can use the HUD in Matchmaker mode to create or connect to instances of your game (also sometimes referred to as "matches" or "sessions") hosted on the internet.

Create Internet Match

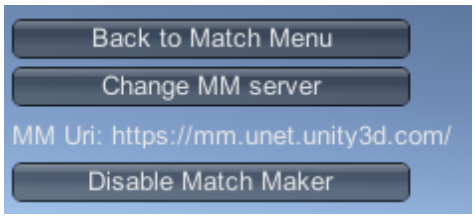
Click **Create Internet Match** to start a new match. Unity's Multiplayer Service creates a new instance of the game (a "match"), which other players can then find and join.

Find Internet Match

Click **Find Internet Match** to send a request to the Unity Multiplayer Service. The Unity Multiplayer Service returns a list of all matches that currently exist for this game.

For example, if two separate players connect, and create a match each named "Match A" and "Match B" respectively, when a third player connects and presses the **Find Internet Match** button, **Match A** and **Match B** are listed as available matches to join.

In the **Network manager** HUD, the available matches appear as a series of buttons, with the text "Join Match: *match name*" (where *match name* is the name chosen by the player who created the match).



An example of results after clicking **Find Internet Match**. In this example, there are no existing matches. If there were, they would appear here and be available for you to join.

To join one of the available matches, click on the “Join Match: *match name*” button for that match. Alternatively, click **Back to Match Menu** to go back to the Matchmaker menu.

When you replace the HUD with your own **UI**, there are better ways to list the available matches. Many multiplayer games display available matches in a scrollable list. You might want to make each entry on the list show the match name, the current and maximum number of players, and other information such as the match mode type, if your decide to make your game have different match modes (such as “capture the flag”, “1 vs 1”, or “cooperative”).

Note: There are some special characters which, if used in a match name, appear modified in the list of available matches in the Network manager HUD. These characters are:

Open square brace: **[**

percent symbol: **%**

Underscore: _

If a match name contains these characters, they are surrounded by square braces in the list of available matches. So a match named “my_game” is listed as “my[_]game”.

Change MM Server

This button is designed for internal use by Unity engineers (for testing the Multiplayer service). It reveals buttons which assign one of three pre-defined URLs to the **MatchMaker Host URI** field in the Network Manager - “local”, “staging” and “internet”. However, the “local” and “staging”** **options are* only intended for internal use by Unity engineers, and not intended for general use*.

If you select the “local” or “staging” options, your game cannot connect to Unity’s Multiuser Service. Therefore you should always make sure this option is set to “internet” (which is the default).

MM Uri Display

This displays the current Matchmaker URI (Uniform Resource Identifier, a string of characters used for identification). To view the URI in the **Inspector**, navigate to the Network Manager **component** and see the **MatchMaker Host URI** field**.** By default this points to the global Unity Multiplayer Service, and for normal multiplayer games using the Unity Multiplayer Service. You should not need to change this. The Unity Multiplayer Service automatically groups the players of your game into regional servers around the world. This grouping ensures fast multiplayer response times between players in the same region, and means that players from Europe, the US, and Asia generally end up playing with other players from their same global region.

If you want to explicitly control which regional server your game connects to, override this value via scripting. For more information and for regional server URIs, see API reference documentation on NetworkMatch.baseUri.

For example, you might want to override the URI if you want to give your players the option of joining a server outside of their global region. If “Player A” in the US wants to connect to a match created via Matchmaker by “Player B” in Europe, they would need to be able to set their desired global region in your game. Therefore you would need to write a UI feature which allows them to select this.

Note: Remember that the Network Manager HUD feature is a temporary aid to development. It allows you to get your multiplayer game running quickly, but you should replace it with your own UI controls when you are ready.

Converting a single-player game to Unity Multiplayer

Leave feedback

This document describes steps to converting a single player game to a multiplayer game, using the new Unity Multiplayer **networking** system. The process described here is a simplified, higher level version of the actual process for a real game; it doesn't always work exactly like this, but it provides a basic recipe for the process.

NetworkManager set-up

Add a new **GameObject** to the **Scene** and rename it "NetworkManager".
Add the **NetworkManager component** to the "NetworkManager" GameObject.
Add the **NetworkManagerHUD** component to the GameObject. This provides the default **UI** for managing the network game state.
See Using the NetworkManager.

Player Prefab set-up

Find the Prefab for the player GameObject in the game, or create a Prefab from the player GameObject
Add the **NetworkIdentity** component to the player Prefab
Check the LocalPlayerAuthority box on the NetworkIdentity
Set the `playerPrefab` in the NetworkManager's **Spawn Info** section to the player Prefab
Remove the player GameObject instance from the Scene if it exists in the Scene
See Player Objects for more information.

Player movement

Add a NetworkTransform component to the player Prefab
Update input and control **scripts** to respect `isLocalPlayer`
Fix **Camera** to use spawned player and `isLocalPlayer`
For example, this script only processes input for the local player:

```
using UnityEngine;
using UnityEngine.Networking;

public class Controls : NetworkBehaviour
{
    void Update()
    {
        if (!isLocalPlayer)
        {
            // exit from update if this is not the local player
            return;
        }

        // handle player input for movement
    }
}
```

Basic player game state

Make scripts that contain important data into NetworkBehaviours instead of MonoBehaviours
Make important member variables into SyncVars
See State Synchronization.

Networked actions

Make scripts that perform important actions into NetworkBehaviours instead of MonoBehaviours
Update functions that perform important player actions to be commands
See Networked Actions.

Non-player GameObjects

Fix non-player prefabs such as enemies:

Add the **NetworkIdentify** component
Add the **NetworkTransform** component
Register spawnable Prefabs with the **NetworkManager**
Update scripts with game state and actions

Spawners

Potentially change spawner scripts to be NetworkBehaviours
Modify spawners to only run on the server (use isServer property or the `OnStartServer()` function)
Call `NetworkServer.Spawn()` for created GameObjects

Spawn positions for players

Add a new GameObject and place it at player's start location
Add the **NetworkStartPosition** component to the new GameObject

Lobby

Create Lobby Scene
Add a new GameObject to the Scene and rename it to "NetworkLobbyManager".
Add the NetworkLobbyManager component to the new GameObject.
Configure the Manager:
Scenes
Prefabs
Spawners

Debugging Information

Leave feedback

Unity provides tools to get information about your game at run time. This information is useful for testing your multiplayer game.

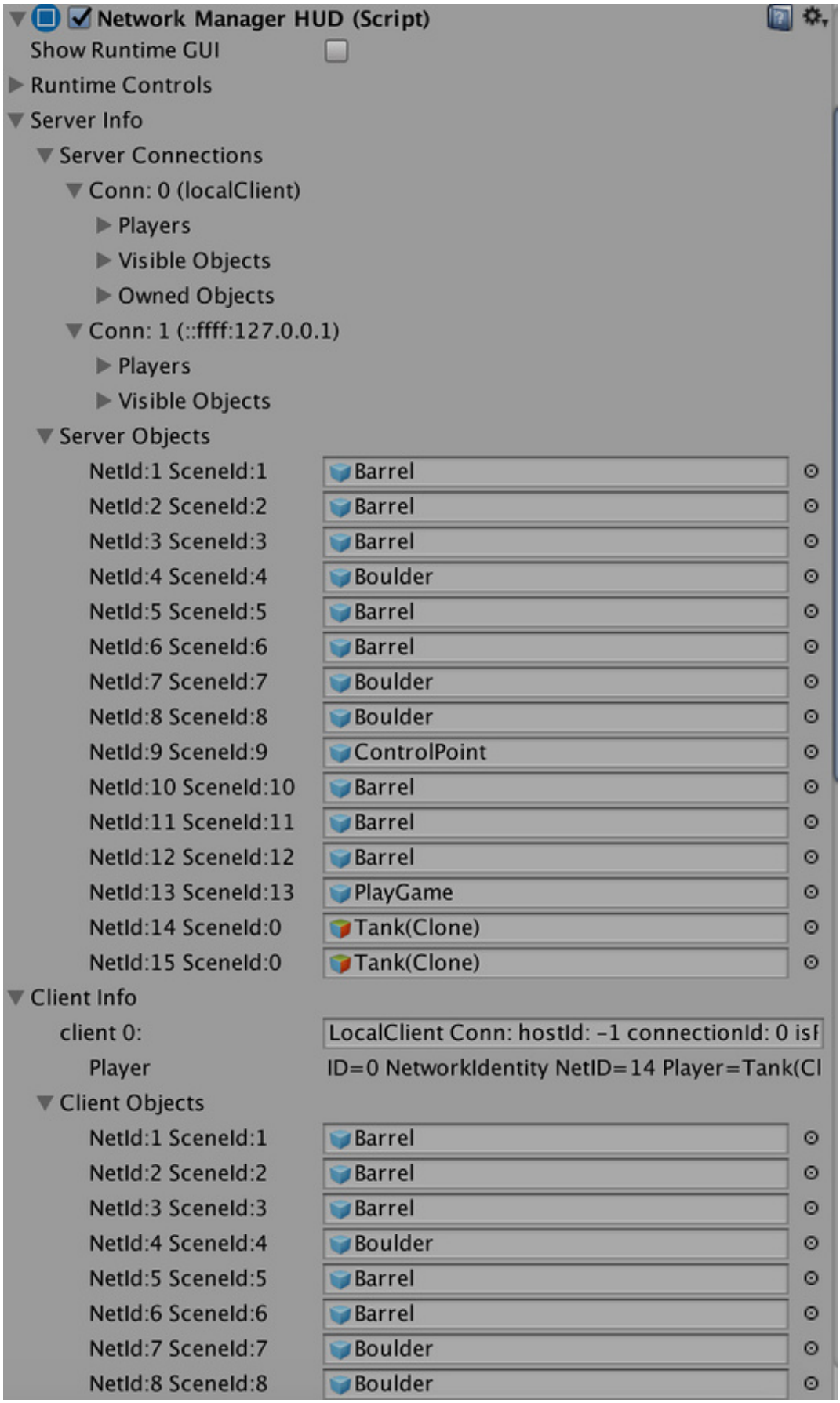
When your game is running in the Editor in Play mode, the **Network Manager** HUD **Inspector** shows additional information about the state of the network at runtime. This includes:

Network connections

Active **GameObjects** on the server which have a Network Identity component

Active GameObjects on the client which have a Network Identity component

Client peers



In Play Mode, the Network Manager HUD component displays additional information about the state of the game and the GameObjects that have spawned.

Additionally, the Network Manager preview pane (at the bottom of the Inspector window) lists the registered message callback handlers.

NetworkManager Message Handlers

Server Message Handlers:

Ready	NetworkManager:OnServerReadyMessageInternal()
Command	NetworkServer:OnCommandMessage()
LocalPlayerTransform	NetworkTransform:HandleTransform()
LocalChildTransform	NetworkTransformChild:HandleChildTransform()
RemovePlayer	NetworkManager:OnServerRemovePlayerMessageInternal()
Animation	NetworkAnimator:OnAnimationServerMessage()
AnimationParams	NetworkAnimator:OnAnimationParametersServerMessage()
AnimationTrigger	NetworkAnimator:OnAnimationTriggerServerMessage()
Fragment	NetworkConnection:OnFragment()
Connect	NetworkManager:OnServerConnectInternal()
Disconnect	NetworkManager:OnServerDisconnectInternal()
AddPlayer	NetworkManager:OnServerAddPlayerMessageInternal()
Error	NetworkManager:OnServerErrorInternal()

Client Message Handlers:

ObjectDestroy	ClientScene:OnLocalClientObjectDestroy()
ObjectHide	ClientScene:OnLocalClientObjectHide()
ObjectSpawn	ClientScene:OnLocalClientObjectSpawn()
ObjectSpawnScene	ClientScene:OnLocalClientObjectSpawnScene()
LocalClientAuthority	ClientScene:OnClientAuthority()
Rpc	ClientScene:OnRPCMessage()
SyncEvent	ClientScene:OnSyncEventMessage()
AnimationTrigger	NetworkAnimator:OnAnimationTriggerClientMessage()
CRC	NetworkClient:OnCRC()
Fragment	NetworkConnection:OnFragment()
Connect	NetworkManager:OnClientConnectInternal()
Disconnect	NetworkManager:OnClientDisconnectInternal()
NotReady	NetworkManager:OnClientNotReadyMessageInternal()
Error	NetworkManager:OnClientErrorInternal()
Scene	NetworkManager:OnClientSceneInternal()

The Network Manager HUD component preview pane, showing registered callback handlers.

The Multiplayer High Level API

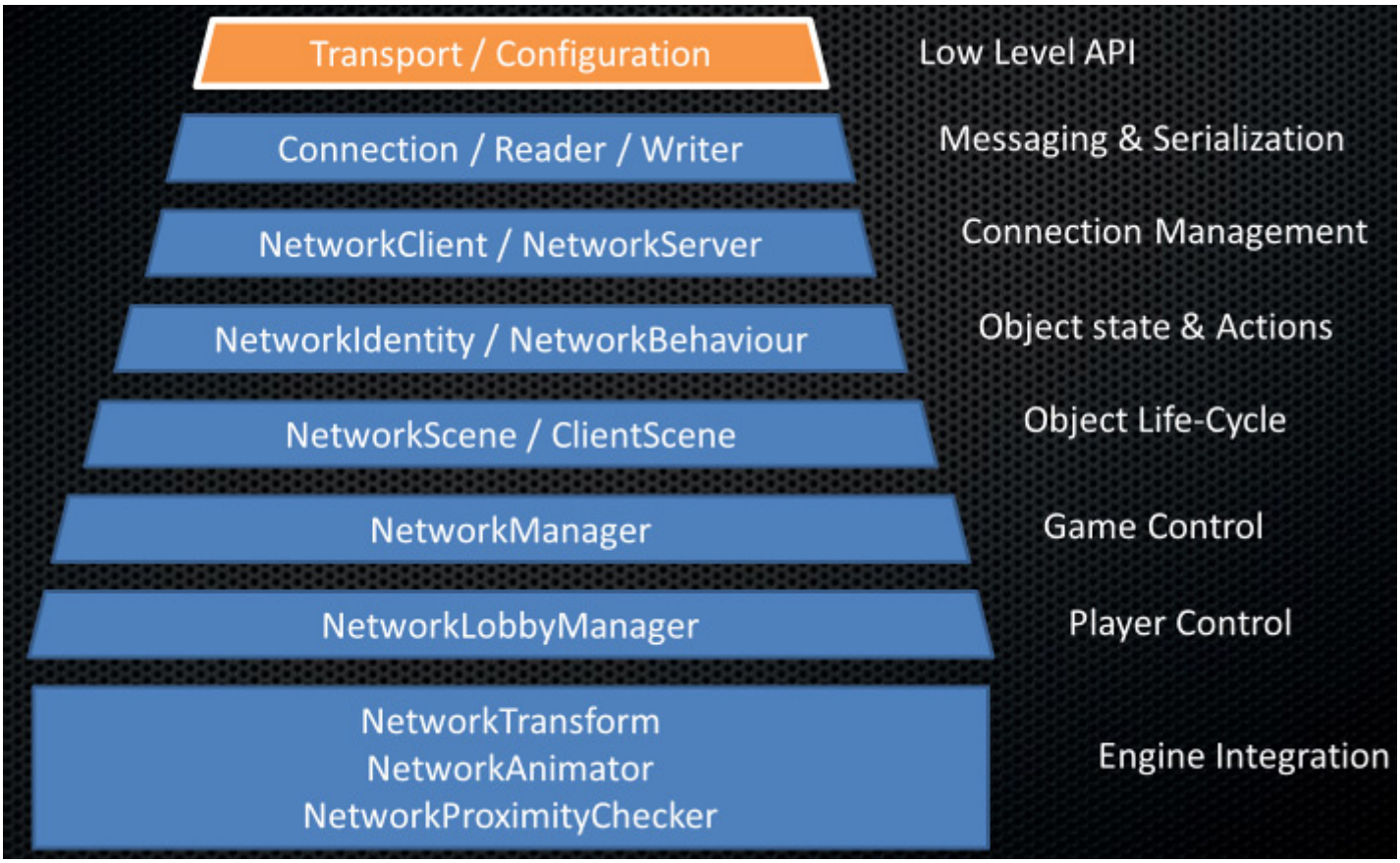
Leave feedback

Unity's multiplayer High Level API (HLAPI) is a system for building multiplayer capabilities for Unity games. It is built on top of the lower level transport real-time communication layer, and handles many of the common tasks that are required for multiplayer games. While the transport layer supports any kind of network topology, the HLAPI is a server authoritative system; although it allows one of the participants to be a client and the server at the same time, so no dedicated server process is required. Working in conjunction with the internet services, this allows multiplayer games to be played over the internet with little work from developers.

The HLAPI is a new set of **networking** commands built into Unity, within a new namespace: **UnityEngine.Networking** . It is focused on ease of use and iterative development and provides services useful for multiplayer games, such as:

Message handlers
General purpose high performance serialization
Distributed object management
State synchronization
Network classes: Server, Client, Connection, etc

The HLAPI is built from a series of layers that add functionality:



This section of the manual explains how to use the multiplayer HLAPI.

Networking HLAPI System Concepts

Leave feedback

Server and Host

In Unity's High Level API (HLAPI) system, multiplayer games include:

A **server**: A server is an instance of the game which all other players connect to when they want to play together. A server often manages various aspects of the game, such as keeping score, and transmit that data back to the client.

Clients: Clients are instances of the game that usually connect from different computers to the server. Clients can connect over a local network, or online.

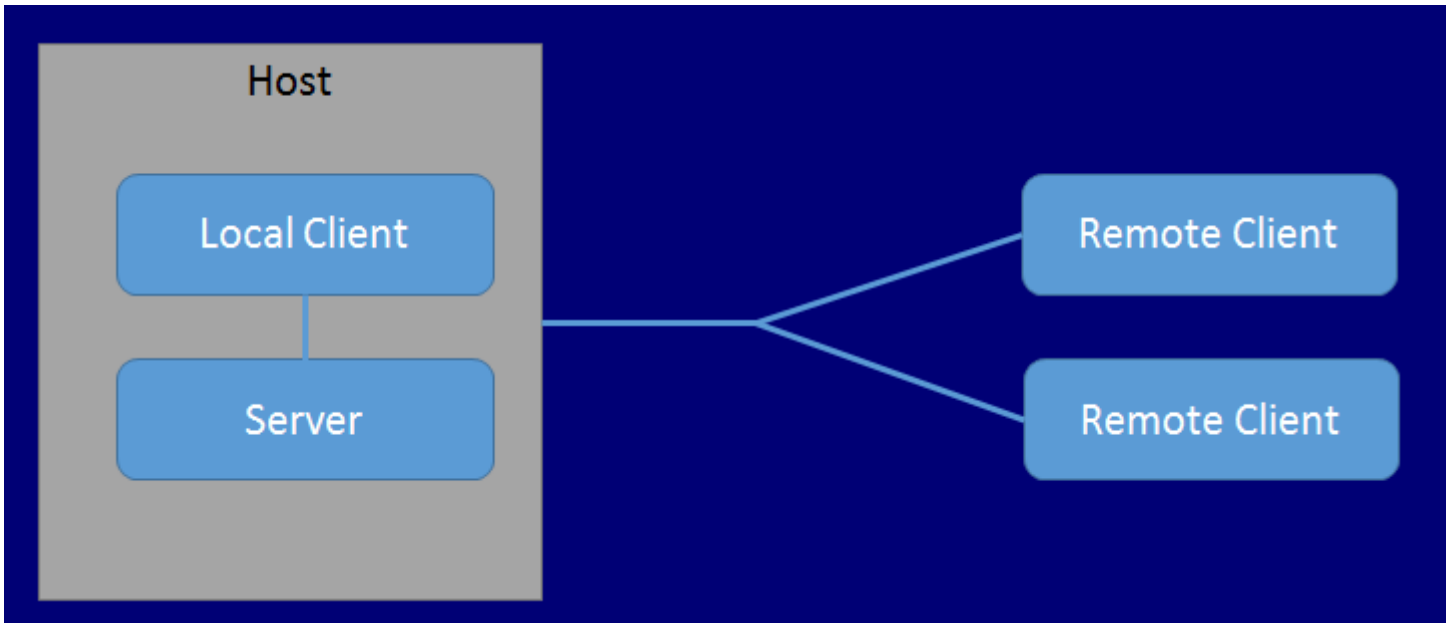
A **client** is an instance of the game that connects to the **server**, so that the person playing it can play the game with other people, who connect on their own clients.

The server might be either a "dedicated server", or a "host server".

Dedicated server: This is an instance of the game that only runs to act as a server.

Host server: When there is no dedicated server, one of the clients also plays the role of the server. This client is the "host server". The host server creates a single instance of the game (called the host), which acts as both server and client.

The diagram below represents three players in a multiplayer game. In this game, one client is also acting as host, which means the client itself is the "local client". The local client connects to the host server, and both run on the same computer. The other two players are remote clients - that is, they are on different computers, connected to the host server.



This diagram shows two remote clients connected to a host.

The **host** is a single instance of your game, acting as both **server** and **client** at the same time. The host uses a special kind of internal client for local client communication, while other clients are remote clients. The local client communicates with the server through direct function calls and message queues, because it is in the same

process. It actually shares the **Scene** with the server. Remote clients communicate with the server over a regular network connection. When you use Unity's HLAPI, this is all handled automatically for you.

One of the aims of the multiplayer system is for the code for local clients and remote clients to be the same, so that you only have to think about one type of client most of the time when developing your game. In most cases, Unity handles this difference automatically, so you should rarely need to think about the difference between your code running on a local client or a remote client.

Instantiate and Spawn

When you make a single player game In Unity, you usually use the `GameObject.Instantiate` method to create new **GameObjects** at runtime. However, with a multiplayer system, the server itself must "spawn" **GameObjects** in order for them to be active within the networked game. When the server spawns **GameObjects**, it triggers the creation of **GameObjects** on connected clients. The spawning system manages the lifecycle of the GameObject, and synchronizes the state of the GameObject based on how you set the GameObject up.

For more details about networked instantiating and spawning, see documentation on Spawning GameObjects.

Players and Local Players

Unity's multiplayer HLAPI system handles player GameObjects differently to non-player GameObjects. When a new player joins the game (when a new client connects to the server), that player's GameObject becomes a "local player" GameObject on the client of that player, and Unity associates the player's connection with the player's GameObject. Unity associates one player GameObject for each person playing the game, and routes networking commands to that individual GameObject. A player cannot invoke a command on another player's GameObject, only their own.

For more details, see documentation on Player GameObjects.

Authority

Servers and clients can both manage a GameObject's behavior. The concept of "authority" refers to how and where a GameObject is managed. Unity's HLAPI is based around "server luthority" as the default state, where the Server (the host) has authority over all GameObjects which do not represent players. Player GameObjects are a special case and treated as having "local authority". You may want to build your game using a different system of authority - for more details, see Network Authority.

Networked GameObjects

Leave feedback

Networked **GameObjects** are **GameObjects** which are controlled and synchronized by Unity's **networking** system. Using synchronized networked **GameObjects**, you can create a shared experience for all the players who are playing an instance of your game. They see and hear the same events and actions - even though that may be from their own unique viewpoints within your game.

Multiplayer games in Unity are typically built using **Scenes** that contain a mix of networked GameObjects and regular (non-networked) GameObjects. The networked GameObjects are those which move or change during gameplay in a way that needs to be synchronized across all users who are playing the game together. Non-networked GameObjects are those which either don't move or change at all during gameplay (for example, static obstacles like rocks or fences), or GameObjects which have movement or changes that don't need to be synchronized across players (for example, a gently swaying tree or clouds passing by in the background of your game).

A networked GameObject is one which has a Network Identity **component** attached. However, a Network Identity **component** alone is not enough for your GameObject to be functional and active in your multiplayer game. The Network Identity **component** is the starting point for synchronization, and it allows the **Network Manager** to synchronize the creation and destruction of the GameObject, but other than that, it does not specify *which properties* of your GameObject should be synchronized.

What exactly should be synchronized on each networked GameObject depends on the type of game you are making, and what each GameObject's purpose is. Some examples of what you might want to synchronize are:

The **position** and **rotation** of moving GameObjects such as the players and non-player characters.

The **animation state** of an animated GameObject

The **value** of a variable, for example how much time is left in the current round of a game, or how much energy a player has.

Some of these things can be automatically synchronized by Unity. The synchronized creation and destruction of networked GameObjects is managed by the NetworkManager, and is known as Spawning. You can use the Network Transform component to synchronize the position and rotation of a GameObject, and you can use the Network Animator component to synchronize the animation of a GameObject.

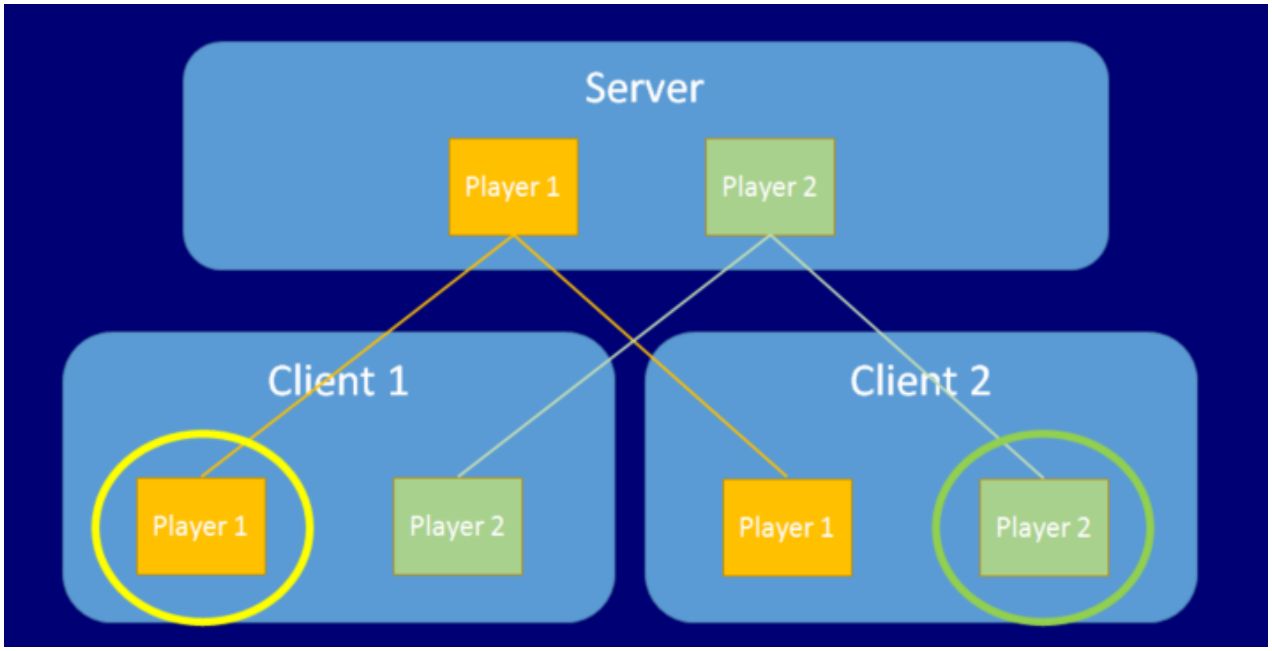
To synchronize other properties of a networked GameObject, you need to use scripting. See State Synchronization for more information about this.

Player GameObjects

Leave feedback

Unity's multiplayer HLAPI system handles player **GameObjects** differently to non-player **GameObjects**. When a new player joins the game (when a new client connects to the server), that player's GameObject becomes a "local player" GameObject on the client of that player, and Unity associates the player's connection with the player's GameObject. Unity associates one player GameObject for each person playing the game, and routes networking commands to that individual GameObject. A player cannot invoke a command on another player's GameObject, only their own.

The NetworkBehaviour class (which you derive from to create your network scripts) has a property called isLocalPlayer. On each client's player GameObject, Unity sets that property to true on the NetworkBehaviour script, and invokes the OnStartLocalPlayer() callback. This means each client has a different GameObject set up like this, because on each client a different GameObject is the one that represents the local player. The diagram below shows two clients and their local players.



In this diagram, the circles represent the player GameObjects marked as the local player on each client

Only the player GameObject that is "yours" (from your point of view as the player) has the `isLocalPlayer`** **flag set. Usually you should set this flag in script to determine whether to process input, whether to make the **camera** track the GameObject, or do any other client-side things that should only occur for the player belonging to that client.

Player GameObjects represent the player (that is, the person playing the game) on the server, and have the ability to run commands from the player's client. These commands are secure client-to-server remote procedure calls. In this server-authoritative system, other non-player server-side GameObjects cannot receive commands directly from client-side GameObjects. This is both for security, and to reduce the complexity of building your game. By routing all incoming commands from users through the player GameObject, you can ensure that these messages come from the right place, the right client, and can be handled in a central location.

The **Network Manager** adds a player every time a client connects to the server. In some situations though, you might want it not to add players until an input event happens - such as a user pressing a "start" button on the controller. To disable automatic player creation, navigate to the **Network Manager component**'s **Inspector** and untick the **Auto Create Player** checkbox.

Custom Player Spawning

Leave feedback

The **Network Manager** offers a built-in simple player spawning feature, however you may want to customize the player spawning process - for example to assign a colour to each new player spawned.

To do this you need to override the default behaviour of the Network Manager with your own script.

When the Network Manager adds a player, it also instantiates a **GameObject** from the Player **Prefab** and associates it with the connection. To do this, the Network Manager calls NetworkServer.AddPlayerForConnection. You can modify this behaviour by overriding NetworkManager.OnServerAddPlayer. The default implementation of `OnServerAddPlayer` instantiates a new player instance from the player Prefab and calls NetworkServer.AddPlayerForConnection to spawn the new player instance. Your custom implementation of `OnServerAddPlayer` must also call `NetworkServer.AddPlayerForConnection`, but your are free to perform any other initialization you require in that method too.

The example below customizes the color of a player. First, add the color script to the player prefab:

```
using UnityEngine;
using UnityEngine.Networking;
class Player : NetworkBehaviour
{
    [SyncVar]
    public Color color;
}
```

Next, create a NetworkManager to handle spawning.

```
using UnityEngine;
using UnityEngine.Networking;
public class MyNetworkManager : NetworkManager
{
    public override void OnServerAddPlayer(NetworkConnection conn, short playerControll
    {
        GameObject player = (GameObject)Instantiate(playerPrefab, Vector3.zero, Quatern
        player.GetComponent<Player>().color = Color.red;
        NetworkServer.AddPlayerForConnection(conn, player, playerControllerId);
    }
}
```

The function `NetworkServer.AddPlayerForConnection` does not have to be called from within `OnServerAddPlayer`. As long as the correct connection object and `playerControllerId` are passed in, it can be called after `OnServerAddPlayer` has returned. This allows asynchronous steps to happen in between, such as loading player data from a remote data source.

Although in most multiplayer games, you typically want one player for each client, the HLAPI treats players and clients as separate concepts.This is because, in some situations (for example, if you have multiple controllers connected to a console system), you might need multiple player GameObjects for a single connection. When there are multiple players on one connection, you should use the `playerControllerId` property to tell them apart. This identifier is scoped to the connection, so that it maps to the ID of the controller associated with the player on that client.

The system automatically spawns the player GameObject passed to `NetworkServer.AddPlayerForConnection` on the server, so you don't need to call NetworkServer.Spawn for the player. Once a player is ready, the active networked GameObjects (that is, GameObjects with an associated NetworkIdentity) in the **Scene** spawn on the player's client. All networked GameObjects in the game are created on that client with their latest state, so they are in sync with the other participants of the game.

You don't need to use playerPrefab on the `NetworkManager` to create player GameObjects. You could use different methods of creating different players.

Ready state

In addition to players, client connections also have a "ready" state. The host sends clients that are ready information about spawned GameObjects and state synchronization updates; clients which are not ready are not sent these updates. When a client initially connects to a server, it is not ready. While in this non-ready state, the client can do things that don't require real-time interactions with the game state on the server, such as loading Scenes, allowing the player to choose an **avatar**, or fill in log-in boxes. Once a client has completed all its pre-game work, and all its **Assets** are loaded, it can call ClientScene.Ready to enter the "ready" state. The simple example above demonstrates implementation of ready states; because adding a player with `NetworkServer.AddPlayerForConnection` also puts the client into the ready state if it is not already in that state.

Clients can send and receive network messages without being ready, which also means they can do so without having an active player GameObject. So a client at a menu or selection screen can connect to the game and interact with it, even though they have no player GameObject. See documentation on Network messages for more details about sending messages without using commands and RPC calls.

Switching players

To replace the player GameObject for a connection, use NetworkServer.ReplacePlayerForConnection. This is useful for restricting the commands that players can issue at certain times, such as in a pre-game lobby screen. This function takes the same arguments as `AddPlayerForConnection`, but allows there to already be a player for that connection. The old player GameObject does not have to be destroyed. The NetworkLobbyManager uses this technique to switch from the NetworkLobbyPlayer GameObject to a gameplay player GameObject when all the players in the lobby are ready.

You can also use `ReplacePlayerForConnection` to respawn a player after their GameObject is destroyed. In some cases it is better to just disable a GameObject and reset its game attributes on respawn. The following code sample demonstrates how to actually replace the destroyed GameObject with a new GameObject:

```
class GameManager
{
    public void PlayerWasKilled(Player player)
    {
        var conn = player.connectionToClient;
        var newPlayer = Instantiate<GameObject>(playerPrefab);
        Destroy(player.gameObject);
        NetworkServer.ReplacePlayerForConnection(conn, newPlayer, 0);
```

```
        }
    }
```

If the player GameObject for a connection is destroyed, then that client cannot execute Commands. They can, however, still send network messages.

To use `ReplacePlayerForConnection` you must have the NetworkConnection GameObject for the player's client to establish the relationship between the GameObject and the client. This is usually the property connectionToClient on the NetworkBehaviour class, but if the old player has already been destroyed, then that might not be readily available.

To find the connection, there are some lists available. If using the `NetworkLobbyManager`, then the lobby players are available in lobbySlots. The NetworkServer also has lists of connections and localConnections.

Spawning GameObjects

Leave feedback

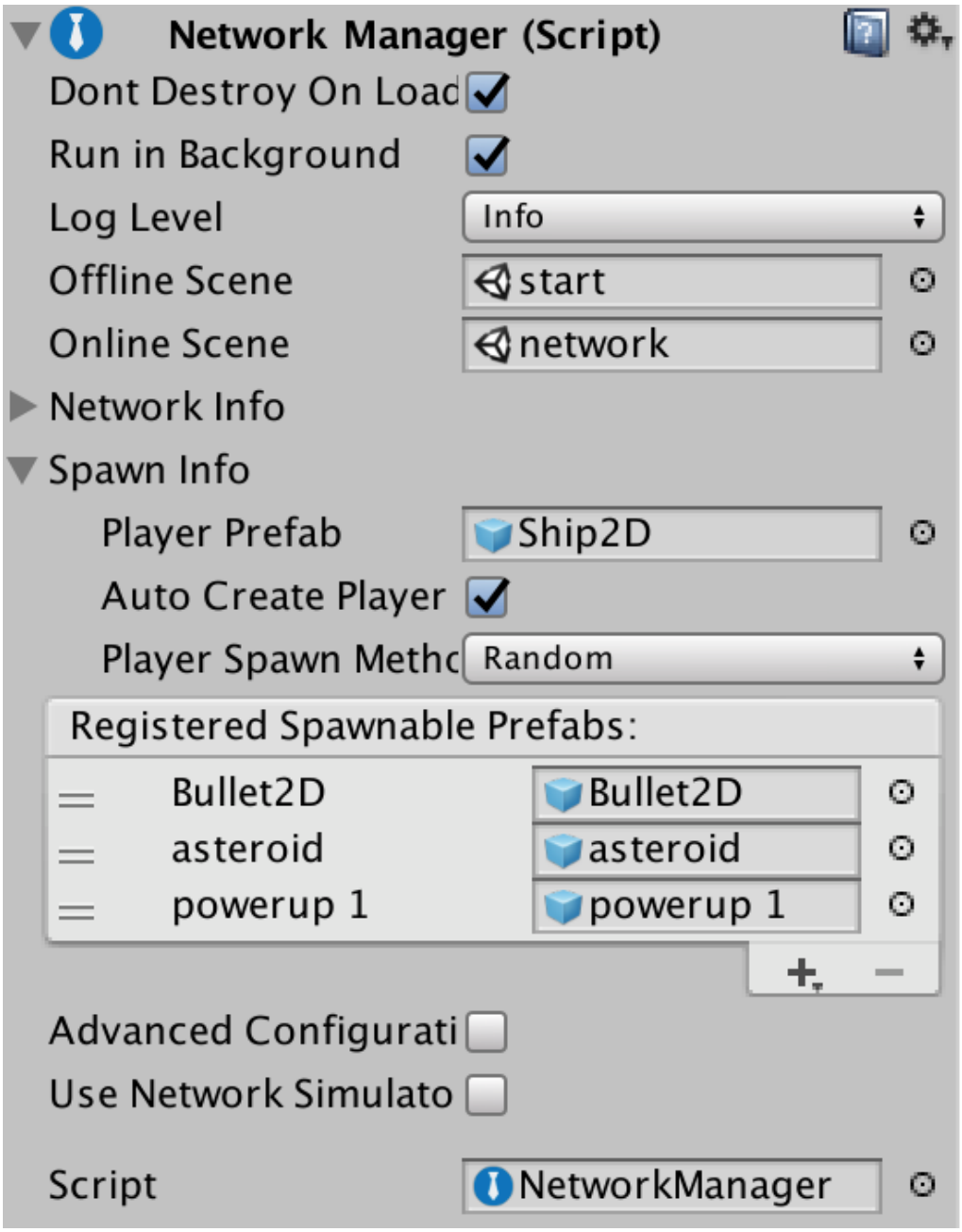
In Unity, you usually “spawn” (that is, create) new **GameObjects** with `Instantiate()`. However, in the multiplayer High Level API, the word “spawn” means something more specific. In the server-authoritative model of the HLAPI, to “spawn” a GameObject on the server means that the GameObject is created on clients connected to the server, and is managed by the spawning system.

Once the GameObject is spawned using this system, state updates are sent to clients whenever the GameObject changes on the server. When Unity destroys the GameObject on the server, it also destroys it on the clients. The server manages spawned GameObjects alongside all other networked GameObjects, so that if another client joins the game later, the server can spawn the GameObjects on that client. These spawned GameObjects have a unique network instance ID called “ **netId**” that is the same on the server and clients for each GameObject. The unique network instance ID is used to route messages set across the network to GameObjects, and to identify GameObjects.

When the server spawns a GameObject with a Network Identity** ****component**, the GameObject spawned on the client has the same “state”. This means it is identical to the GameObject on the server; it has the same Transform, movement state, and (if NetworkTransform and SyncVars are used) synchronized variables. Therefore, client GameObjects are always up-to-date when Unity creates them. This avoids issues such as GameObjects spawning at the wrong initial location, then reappearing at their correct position when a state update arrives.

The **Network Manager** can only spawn and synchronize GameObjects from registered Prefabs, so you must register the specific GameObject Prefabs with the **Network Manager** that you want to be able to spawn during your game. The **Network Manager** will only accept GameObject Prefabs which have a Network Identity component attached, so you must make sure you add a Network Identity component to your **Prefab** before trying to register it with the **Network Manager**.

To register a Prefab with the Network Manager in the Editor, select the Network Manager GameObject, and in the **Inspector**, navigate to the Network Manager component. Click the triangle next to **Spawn Info** to open the settings, then under **Registered Spawnable Prefabs**, click the plus (**+**) button. Drag and drop Prefabs into the empty field to assign them to the list.



The Network Manager Inspector with the *Spawn Info** foldout expanded, displaying three assigned Prefabs under *Registered Spawnable Prefabs*

Spawning without the Network Manager

For more advanced users, you may find that you want to register Prefabs and spawn GameObjects without using the NetworkManager component.

To spawn GameObjects without using the Network Manager, you can handle the Prefab registration yourself via script. Use the ClientScene.RegisterPrefab method to register Prefabs to the Network Manager.

Example: MyNetworkManager

```
using UnityEngine;
using UnityEngine.Networking;

public class MyNetworkManager : MonoBehaviour
```

```
{
    public GameObject treePrefab;
    NetworkClient myClient;

    // Create a client and connect to the server port
    public void ClientConnect() {
        ClientScene.RegisterPrefab(treePrefab);
        myClient = new NetworkClient();
        myClient.RegisterHandler(MsgType.Connect, OnClientConnect);
        myClient.Connect("127.0.0.1", 4444);
    }

    void OnClientConnect(NetworkMessage msg) {
        Debug.Log("Connected to server: " + msg.conn);
    }
}
```

In this example, you create an empty GameObject to act as the Network Manager, then create and attach the `MyNetworkManager` script (above) to that GameObject. Create a Prefab that has a Network Identity component attached to it, and drag that onto the **treePrefab** slot on the MyNetworkManager component in the Inspector. This ensures that when the server spawns the tree GameObject, it also creates the same kind of GameObject on the clients.

Registering Prefabs ensures that the **Asset** is loaded with the **Scene**, so that there is no stalling or loading time for creating the **Asset**.

However, for the script to work, you also need to add code for the server. Add this to the **MyNetworkManager** script:

```
public void ServerListen() {
    NetworkServer.RegisterHandler(MsgType.Connect, OnServerConnect);
    NetworkServer.RegisterHandler(MsgType.Ready, OnClientReady);
    if (NetworkServer.Listen(4444))
        Debug.Log("Server started listening on port 4444");
}

// When client is ready spawn a few trees
void OnClientReady(NetworkMessage msg) {
    Debug.Log("Client is ready to start: " + msg.conn);
    NetworkServer.SetClientReady(msg.conn);
    SpawnTrees();
}

void SpawnTrees() {
    int x = 0;
    for (int i = 0; i < 5; ++i) {
        var treeGo = Instantiate(treePrefab, new Vector3(x++, 0, 0), Quaternion.identit
        NetworkServer.Spawn(treeGo);
    }
```

```
}

void OnServerConnect(NetworkMessage msg) {
    Debug.Log("New client connected: " + msg.conn);
}
```

The server does not need to register anything, as it knows what GameObject is being spawned (and the asset ID is sent in the spawn message). The client needs to be able to look up the GameObject, so it must be registered on the client.

When writing your own network manager, it's important to make the client ready to receive state updates before calling the spawn command on the server, otherwise they won't be sent. If you're using Unity's built-in Network Manager component, this happens automatically.

For more advanced uses, such as object pools or dynamically created Assets, you can use the ClientScene.RegisterSpawnHandler method, which allows callback functions to be registered for client-side spawning. See documentation on Custom Spawn Functions for an example of this.

If the GameObject has a network state like synchronized variables, then that state is synchronized with the spawn message. In the following example, this script is attached to the tree Prefab:

```
using UnityEngine;
using UnityEngine.Networking;
class Tree : NetworkBehaviour {
    [SyncVar]
    public int numLeaves;
    public override void OnStartClient() {
        Debug.Log("Tree spawned with leaf count " + numLeaves);
    }
}
```

With this script attached, you can change the `numLeaves` variable and modify the `SpawnTrees` function to see it accurately reflected on the client:

```
void SpawnTrees() {
    int x = 0;
    for (int i = 0; i < 5; ++i) {
        var treeGo = Instantiate(treePrefab, new Vector3(x++, 0, 0), Quaternion.identit
        var tree = treeGo.GetComponent<Tree>();
        tree.numLeaves = Random.Range(10,200);
        Debug.Log("Spawning leaf with leaf count " + tree.numLeaves);
        NetworkServer.Spawn(treeGo);
    }
}
```

Attach the `Tree` script to the `treePrefab` script created earlier to see this in action.

Constraints

A **NetworkIdentity** must be on the root GameObject of a spawnable Prefab. Without this, the Network Manager can't register the Prefab.

NetworkBehaviour **scripts** must be on the same GameObject as the NetworkIdentity, not on child GameObjects

GameObject creation flow

The actual flow of internal operations that takes place for spawning GameObjects is:

Prefab with Network Identity component is registered as spawnable.

GameObject is instantiated from the Prefab on the server.

Game code sets initial values on the instance (note that 3D physics forces applied here do not take effect immediately).

`NetworkServer.Spawn()` is called with the instance.

The state of the SyncVars on the instance on the server are collected by calling `OnSerialize()` on Network Behaviour components.

A network message of type `MsgType.ObjectSpawn` is sent to connected clients that includes the SyncVar data.

`OnStartServer()` is called on the instance on the server, and `isServer` is set to `true`

Clients receive the `ObjectSpawn` message and create a new instance from the registered Prefab.

The SyncVar data is applied to the new instance on the client by calling OnDeserialize() on Network Behaviour components.

`OnStartClient()` is called on the instance on each client, and `isClient` is set to `true`

As gameplay progresses, changes to SyncVar values are automatically synchronized to clients. This continues until game ends.

`NetworkServer.Destroy()` is called on the instance on the server.

A network message of type `MsgType.ObjectDestroy` is sent to clients.

`OnNetworkDestroy()` is called on the instance on clients, then the instance is destroyed.

Player GameObjects

Player GameObjects in the HLAPI work slightly differently to non-player GameObjects. The flow for spawning player GameObjects with the Network Manager is:

Prefab with `NetworkIdentity` is registered as the `PlayerPrefab`

Client connects to the server

Client calls `AddPlayer()`, network message of type `MsgType.AddPlayer` is sent to the server

Server receives message and calls `NetworkManager.OnServerAddPlayer()`

GameObject is instantiated from the PlayerPrefab on the server

`NetworkManager.AddPlayerForConnection()` is called with the new player instance on the server

The player instance is spawned - you do not have to call `NetworkServer.Spawn()` for the player instance. The spawn message is sent to all clients like on a normal spawn.

A network message of type `MsgType.Owner` is sent to the client that added the player (only that client!)

The original client receives the network message

`OnStartLocalPlayer()` is called on the player instance on the original client, and `isLocalPlayer` is set to true

Note that `OnStartLocalPlayer()` is called after `OnStartClient()`, because it only happens when the ownership message arrives from the server after the player GameObject is spawned, so `isLocalPlayer` is not set in `OnStartClient()`.

Because `OnStartLocalPlayer` is only called for the client's local player GameObject, it is a good place to perform initialization that should only be done for the local player. This could include enabling input processing, and enabling **camera** tracking for the player GameObject.

Spawning GameObjects with client authority

To spawn GameObjects and assign authority of those GameObjects to a particular client, use NetworkServer.SpawnWithClientAuthority, which takes as an argument the `NetworkConnection` of the client that is to be made the authority.

For these GameObjects, the property `hasAuthority` is true on the client with authority, and `OnStartAuthority()` is called on the client with authority. That client can issue commands for that GameObject. On other clients (and on the host), `hasAuthority` is false.

Objects spawned with client authority must have `LocalPlayerAuthority` set in their `NetworkIdentity`.

For example, the tree spawn example above can be modified to allow the tree to have client authority like this (note that we now need to pass in a NetworkConnection GameObject for the owning client's connection):

```
void SpawnTrees(NetworkConnection conn) {
    int x = 0;
    for (int i = 0; i < 5; ++i)
    {
        var treeGo = Instantiate(treePrefab, new Vector3(x++, 0, 0), Quaternion.identit
        var tree = treeGo.GetComponent<Tree>();
        tree.numLeaves = Random.Range(10,200);
        Debug.Log("Spawning leaf with leaf count " + tree.numLeaves);
        NetworkServer.SpawnWithClientAuthority(treeGo, conn);
    }
}
```

The Tree script can now be modified to send a command to the server:

```
public override void OnStartAuthority() {
    CmdMessageFromTree("Tree with " + numLeaves + " reporting in");
}

[Command]
void CmdMessageFromTree(string msg) {
    Debug.Log("Client sent a tree message: " + msg);
}
```

Note that you can't just add the `CmdMessageFromTree` call into `OnStartClient`, because at that point the authority has not been set yet, so the call would fail.

Custom Spawn Functions

Leave feedback

You can use spawn handler functions to customize the default behaviour when creating spawned **GameObjects** from **prefabs** on the client. Spawn handler functions ensure you have full control of how you spawn the GameObject, as well as how you destroy it.

Use ClientScene.RegisterSpawnHandler to register functions to spawn and destroy client GameObjects. The server creates GameObjects directly, and then spawns them on the clients through this functionality. This function takes the **asset** ID of the GameObject and two function delegates: one to handle creating GameObjects on the client, and one to handle destroying GameObjects on the client. The **asset** ID can be a dynamic one, or just the **asset** ID found on the prefab GameObject you want to spawn (if you have one).

The spawn / un-spawner need to have this GameObject signature. This is defined in the high level API.

```
// Handles requests to spawn GameObjects on the client
public delegate GameObject SpawnDelegate(Vector3 position, NetworkHash128 assetId);

// Handles requests to unspawn GameObjects on the client
public delegate void UnSpawnDelegate(GameObject spawned);
```

The asset ID passed to the spawn function can be found on NetworkIdentity.assetId for prefabs, where it is populated automatically. The registration for a dynamic asset ID is handled like this:

```
// generate a new unique assetId
NetworkHash128 creatureAssetId = NetworkHash128.Parse("e2656f");

// register handlers for the new assetId
ClientScene.RegisterSpawnHandler(creatureAssetId, SpawnCreature, UnSpawnCreature);

// get assetId on an existing prefab
NetworkHash128 coinAssetId = coinPrefab.GetComponent<NetworkIdentity>().assetId;

// register handlers for an existing prefab you'd like to custom spawn
ClientScene.RegisterSpawnHandler(coinAssetId, SpawnCoin, UnSpawnCoin);

// spawn a coin - SpawnCoin is called on client
NetworkServer.Spawn(gameObject, coinAssetId);
```

The spawn functions themselves are implemented with the delegate signature. Here is the coin spawner. The SpawnCreature would look the same, but have different spawn logic:

```
public GameObject SpawnCoin(Vector3 position, NetworkHash128 assetId)
{
    return (GameObject)Instantiate(m_CoinPrefab, position, Quaternion.identity);
```

```
}
public void UnSpawnCoin(GameObject spawned)
{
    Destroy(spawned);
}
```

When using custom spawn functions, it is sometimes useful to be able to unspawn GameObjects without destroying them. This can be done by calling NetworkServer.UnSpawn. This causes a message to be sent to clients to un-spawn the GameObject, so that the custom un-spawn function will be called on the clients. The GameObject is not destroyed when this function is called.

Note that on the host, GameObjects are not spawned for the local client, because they already exist on the server. This also means that no spawn handler functions are called.

Setting up a GameObject pool with custom spawn handlers

Here is an example of how you might set up a very simple GameObject pooling system with custom spawn handlers. Spawning and unspawning then puts GameObjects in or out of the pool.

```
using UnityEngine;
using UnityEngine.Networking;
using System.Collections;

public class SpawnManager : MonoBehaviour
{
    public int m_ObjectPoolSize = 5;
    public GameObject m_Prefab;
    public GameObject[] m_Pool;

    public NetworkHash128 assetId { get; set; }

    public delegate GameObject SpawnDelegate(Vector3 position, NetworkHash128 assetId);
    public delegate void UnSpawnDelegate(GameObject spawned);

    void Start()
    {
        assetId = m_Prefab.GetComponent<NetworkIdentity> ().assetId;
        m_Pool = new GameObject[m_ObjectPoolSize];
        for (int i = 0; i < m_ObjectPoolSize; ++i)
        {
            m_Pool[i] = (GameObject)Instantiate(m_Prefab, Vector3.zero, Quaternion.identity);
            m_Pool[i].name = "PoolObject" + i;
            m_Pool[i].SetActive(false);
        }

        ClientScene.RegisterSpawnHandler(assetId, SpawnObject, UnSpawnObject);
    }

    public GameObject GetFromPool(Vector3 position)
    {
        foreach (var obj in m_Pool)
```

```
        {
            if (!obj.activeInHierarchy)
            {
                Debug.Log("Activating GameObject " + obj.name + " at " + position);
                obj.transform.position = position;
                obj.SetActive (true);
                return obj;
            }
        }
        Debug.LogError ("Could not grab GameObject from pool, nothing available");
        return null;
    }

    public GameObject SpawnObject(Vector3 position, NetworkHash128 assetId)
    {
        return GetFromPool(position);
    }

    public void UnSpawnObject(GameObject spawned)
    {
        Debug.Log ("Re-pooling GameObject " + spawned.name);
        spawned.SetActive (false);
    }
}
```

To use this manager, create a new empty GameObject and name it “SpawnManager”. Create a new script called *SpawnManager,* copy in the code sample above, and attach it to the* *new SpawnManager GameObject. Next, drag a prefab you want to spawn multiple times to the **P refab** field, and set the **Object Pool Size** (default is 5).

Finally, set up a reference to the SpawnManager in the script you are using for player movement:

```
SpawnManager spawnManager;

void Start()
{
    spawnManager = GameObject.Find("SpawnManager").GetComponent<SpawnManager> ();
}
```

Your player logic might contain something like this, which moves and fires coins:

```
void Update()
{
    if (!isLocalPlayer)
        return;
```

```
    var x = Input.GetAxis("Horizontal")*0.1f;
    var z = Input.GetAxis("Vertical")*0.1f;

    transform.Translate(x, 0, z);

    if (Input.GetKeyDown(KeyCode.Space))
    {
        // Command function is called on the client, but invoked on the server
        CmdFire();
    }
}
```

In the fire logic on the player, make it use the GameObject pool:

```
[Command]
void CmdFire()
{
    // Set up coin on server
    var coin = spawnManager.GetFromPool(transform.position + transform.forward);
    coin.GetComponent<Rigidbody>().velocity = transform.forward*4;

    // spawn coin on client, custom spawn handler is called
    NetworkServer.Spawn(coin, spawnManager.assetId);

    // when the coin is destroyed on the server, it is automatically destroyed on clients
    StartCoroutine (Destroy (coin, 2.0f));
}

public IEnumerator Destroy(GameObject go, float timer)
{
    yield return new WaitForSeconds (timer);
    spawnManager.UnSpawnObject(go);
    NetworkServer.UnSpawn(go);
}
```

The automatic destruction shows how the GameObjects are returned to the pool and re-used when you fire again.

Network Authority

Leave feedback

Servers and clients can both manage a **GameObject**'s behavior. The concept of "authority" refers to how and where a **GameObject** is managed.

Server Authority

The default state of authority in Unity networked games using the HLAPI is that the Server has authority over all GameObjects which do not represent players. This means - for example - the server would manage control of all collectable items, moving platforms, NPCs - and any other parts of your game that players can interac and player GameObjects have authority on their owner's client (meaning the client manages their behavior).

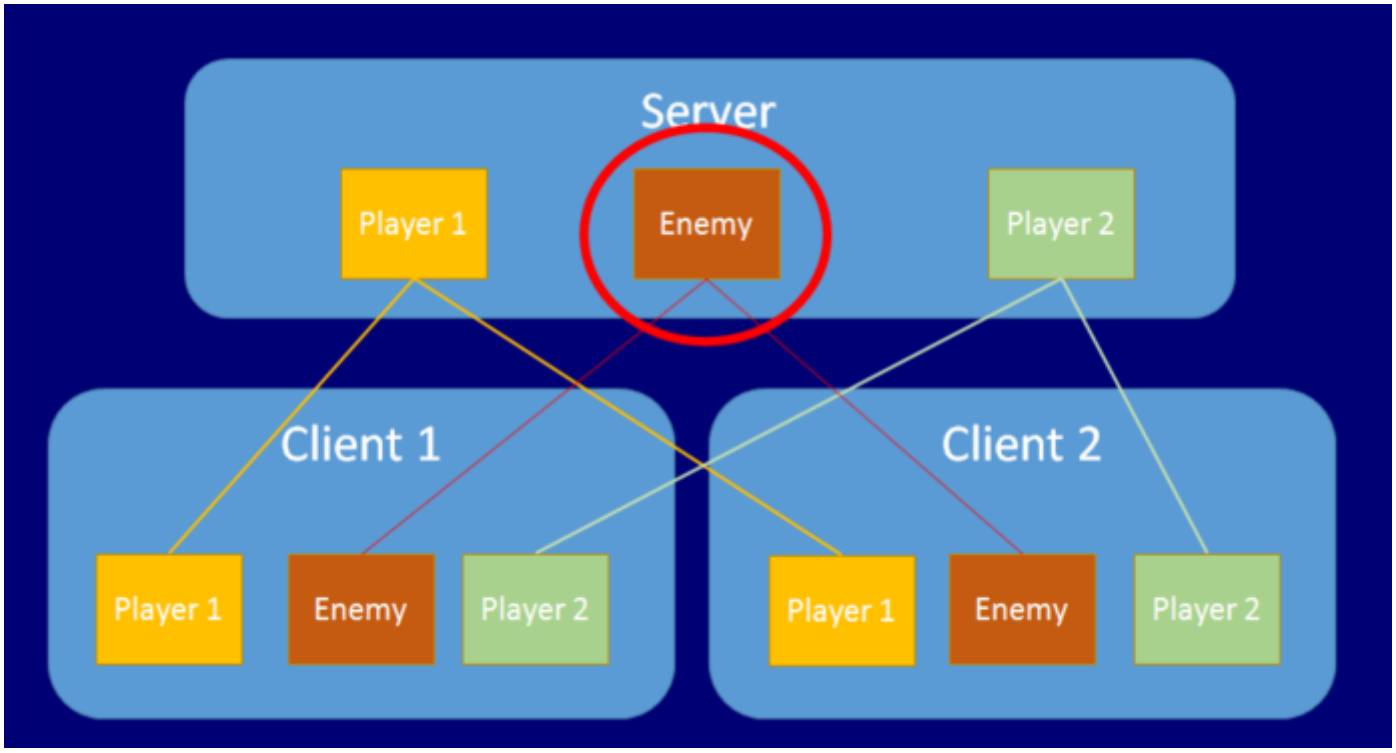
Local authority

Local authority (sometimes referred to as client authority) means the local client has authoritative control over a particular networked GameObject. This is in contrast to the default state which is that the server has authoritative control over networked GameObjects.

In addition to `isLocalPlayer`**, you can choose to make the player GameObjects have "local authority". This means that the player GameObject on its owner's client is responsible for (or has authority over) itself. This is particularly useful for controlling movement; it means that each client has authority over how their own player GameObject is being controlled.

To enable local player authority on a GameObject, tick the Network Identity **component**'s **Local Player Authority** checkbox. The Network Transform component uses this "authority" setting, and sends movement information from the client to the other clients if this is set.

See Scripting API Reference documentation on NetworkIdentity and localPlayerAuthority for information on implementing local player authority via script.



This image shows the Enemy object under server authority. The enemy appears on Client 1 and Client 2, but the server is in charge of its position, movement, and behavior

Use the NetworkIdentity.hasAuthority property to find out whether a GameObject has local authority (also accessible on `NetworkBehaviour` for convenience). Non-player GameObjects have authority on the server, and player GameObjects with **localPlayerAuthority** set have authority on their owner's client.

Local (Client) Authority for Non-Player GameObjects

It is possible to have client authority over non-player GameObjects. There are two ways to do this. One is to spawn the GameObject using NetworkServer.SpawnWithClientAuthority, and pass the network connection of the client to take ownership. The other is to use NetworkIdentity.AssignClientAuthority with the network connection of the client to take ownership.

Assigning authority to a client causes Unity to call OnStartAuthority() on each `NetworkBehaviour` on the GameObject, and sets the `hasAuthority`** **property to true. On other clients, the** `hasAuthority` **property remains false. Non-player GameObjects with client authority can send commands, just like players can. These commands are run on the server instance of the GameObject, not on the player associated with the connection.

If you want non-player GameObjects to have client authority, you must enable **localPlayerAuthority** on their Network Identity component. The example below spawns a GameObject and assigns authority to the client of the player that spawned it.

```
[Command]
void CmdSpawn()
{
    var go = (GameObject)Instantiate(otherPrefab, transform.position + new Vecto
    NetworkServer.SpawnWithClientAuthority(go, connectionToClient);
}
```

Network Context Properties

The `NetworkBehaviour` class contains properties that allow **scripts** to know what the context of a networked GameObject is at any time.

isServer - true if the GameObject is on a server (or host) and has been spawned.

isClient - true if the GameObject is on a client, and was created by the server.

isLocalPlayer - true if the GameObject is a player GameObject for this client.

hasAuthority - true if the GameObject is owned by the local process

To see these properties, select the GameObject you want to inspect, and in the **Inspector** window, view the preview window for the NetworkBehaviour scripting components. You can use the value of these properties to execute code based on the context in which the script is running.

State synchronization

Leave feedback

State synchronization refers to the synchronization of values such as integers, floating point numbers, strings and boolean values belonging to **scripts** on your networked **GameObjects**.

State synchronization is done from the Server to remote clients. The *local* client does not have data serialized to it. It does not need it, because it shares the **Scene** with the server. However, SyncVar hooks are called on local clients.

Data is not synchronized in the opposite direction - from remote clients to the server. To do this, you need to use Commands.

SyncVars

SyncVars are variables of scripts that inherit from NetworkBehaviour, which are synchronized from the server to clients. When a GameObject is spawned, or a new player joins a game in progress, they are sent the latest state of all SyncVars on networked objects that are visible to them. Use the `[SyncVar]` custom attribute to specify which variables in your script you want to synchronize, like this:

class Player : NetworkBehaviour {

```
[SyncVar]
int health;

public void TakeDamage(int amount)
{
    if (!isServer)
        return;

    health -= amount;
}
```

}

The state of SyncVars is applied to GameObjects on clients before OnStartClient() is called, so the state of the object is always up-to-date inside OnStartClient().

SyncVars can be basic types such as integers, strings and floats. They can also be Unity types such as Vector3 and user-defined structs, but updates for struct SyncVars are sent as monolithic updates, not incremental changes if fields within a struct change. You can have up to 32 SyncVars on a single NetworkBehaviour script, including SyncLists (see next section, below).

The server automatically sends SyncVar updates when the value of a SyncVar changes, so you do not need to track when they change or send information about the changes yourself.

SyncLists

While SyncVars contain values, SyncLists contain lists of values. SyncList contents are included in initial state updates along with SyncVar states. Since SyncList is a class which synchronises its own contents, SyncLists do not require the SyncVar attribute. The following types of SyncList are available for basic types:

SyncListString

SyncListFloat

SyncListInt

SyncListUInt

SyncListBool

There is also SyncListStruct, which you can use to synchronize lists of your own struct types. When using SyncListStruct, the struct type that you choose to use can contain members of basic types, arrays, and common Unity types. They cannot contain complex classes or generic containers, and only public variables in these structs are serialized.

SyncLists have a SyncListChanged delegate named Callback that allows clients to be notified when the contents of the list change. This delegate is called with the type of operation that occurred, and the index of the item that the operation was for.

```
public class MyScript : NetworkBehaviour
{
    public struct Buf
    {
        public int id;
        public string name;
        public float timer;
    };

    public class TestBufs : SyncListStruct<Buf> {}
    TestBufs m_bufs = new TestBufs();

    void BufChanged(Operation op, int itemIndex)
    {
        Debug.Log("buf changed:" + op);
    }

    void Start()
    {
        m_bufs.Callback = BufChanged;
    }
}
```

Advanced State Synchronization

Leave feedback

In most cases, the use of SyncVars is enough for your game **scripts** to serialize their state to clients. However in some cases you might require more complex serialization code. This page is only relevant for advanced developers who need customized synchronization solutions that go beyond Unity's normal SyncVar feature.

Custom Serialization Functions

To perform your own custom serialization, you can implement virtual functions on NetworkBehaviour to be used for SyncVar serialization. These functions are:

```
public virtual bool OnSerialize(NetworkWriter writer, bool initialState);
```

```
public virtual void OnDeSerialize(NetworkReader reader, bool initialState);
```

Use the `initialState` flag to differentiate between the first time a **GameObject** is serialized and when incremental updates can be sent. The first time a **GameObject** is sent to a client, it must include a full state snapshot, but subsequent updates can save on bandwidth by including only incremental changes. Note that SyncVar hook functions are not called when `initialState` is true; they are only called for incremental updates.

If a class has SyncVars, then implementations of these functions are added automatically to the class, meaning that a class that has SyncVars cannot also have custom serialization functions.

The `OnSerialize` function should return true to indicate that an update should be sent. If it returns true, the dirty bits for that script are set to zero. If it returns false, the dirty bits are not changed. This allows multiple changes to a script to be accumulated over time and sent when the system is ready, instead of every frame.

Serialization Flow

GameObjects with the Network Identity **component** attached can have multiple scripts derived from `NetworkBehaviour`. The flow for serializing these GameObjects is:

On the server:

Each `NetworkBehaviour` has a dirty mask. This mask is available inside `OnSerialize` as `syncVarDirtyBits`

Each SyncVar in a `NetworkBehaviour` script is assigned a bit in the dirty mask.

Changing the value of SyncVars causes the bit for that SyncVar to be set in the dirty mask

Alternatively, calling `SetDirtyBit()` writes directly to the dirty mask

NetworkIdentity GameObjects are checked on the server as part of it's update loop

If any `NetworkBehaviours` on a `NetworkIdentity` are dirty, then an `UpdateVars` packet is created for that GameObject

The `UpdateVars` packet is populated by calling `OnSerialize` on each `NetworkBehaviour` on the GameObject

`NetworkBehaviours` that are not dirty write a zero to the packet for their dirty bits

`NetworkBehaviours` that are dirty write their dirty mask, then the values for the SyncVars that have changed

If `OnSerialize` returns true for a `NetworkBehaviour`, the dirty mask is reset for that `NetworkBehaviour`, so it does not send again until its value changes.

The `UpdateVars` packet is sent to ready clients that are observing the GameObject

On the client:

an `UpdateVars packet` is received for a GameObject

The `OnDeserialize` function is called for each `NetworkBehaviour` script on the GameObject

Each `NetworkBehaviour` script on the GameObject reads a dirty mask.

If the dirty mask for a `NetworkBehaviour` is zero, the `OnDeserialize` function returns without reading any more

If the dirty mask is non-zero value, then the `OnDeserialize` function reads the values for the SyncVars that correspond to the dirty bits that are set

If there are SyncVar hook functions, those are invoked with the value read from the stream.

So for this script:

```
public class data : NetworkBehaviour
{

    [SyncVar]
    public int int1 = 66;

    [SyncVar]
    public int int2 = 23487;

    [SyncVar]
    public string MyString = "Example string";
```

```
}
```

The following code sample demonstrates the generated `OnSerialize` function:

```
public override bool OnSerialize(NetworkWriter writer, bool forceAll)
{
    if (forceAll)
    {
        // The first time a GameObject is sent to a client, send all the data (a
        writer.WritePackedUInt32((uint)this.int1);
        writer.WritePackedUInt32((uint)this.int2);
        writer.Write(this.MyString);
        return true;
    }
    bool wroteSyncVar = false;
    if ((base.get_syncVarDirtyBits() & 1u) != 0u)
    {
        if (!wroteSyncVar)
        {
            // Write dirty bits if this is the first SyncVar written
            writer.WritePackedUInt32(base.get_syncVarDirtyBits());
            wroteSyncVar = true;
        }
        writer.WritePackedUInt32((uint)this.int1);
    }
    if ((base.get_syncVarDirtyBits() & 2u) != 0u)
    {
        if (!wroteSyncVar)
        {
            // Write dirty bits if this is the first SyncVar written
            writer.WritePackedUInt32(base.get_syncVarDirtyBits());
            wroteSyncVar = true;
        }
        writer.WritePackedUInt32((uint)this.int2);
    }
    if ((base.get_syncVarDirtyBits() & 4u) != 0u)
    {
        if (!wroteSyncVar)
        {
            // Write dirty bits if this is the first SyncVar written
            writer.WritePackedUInt32(base.get_syncVarDirtyBits());
```

```
            wroteSyncVar = true;
        }
        writer.Write(this.MyString);
    }

    if (!wroteSyncVar)
    {
        // Write zero dirty bits if no SyncVars were written
        writer.WritePackedUInt32(0);
    }
    return wroteSyncVar;
}
```

The following code sample demonstrates the `OnDeserialize` function:

```
public override void OnDeserialize(NetworkReader reader, bool initialState)
{
    if (initialState)
    {
        this.int1 = (int)reader.ReadPackedUInt32();
        this.int2 = (int)reader.ReadPackedUInt32();
        this.MyString = reader.ReadString();
        return;
    }
    int num = (int)reader.ReadPackedUInt32();
    if ((num & 1) != 0)
    {
        this.int1 = (int)reader.ReadPackedUInt32();
    }
    if ((num & 2) != 0)
    {
        this.int2 = (int)reader.ReadPackedUInt32();
    }
    if ((num & 4) != 0)
    {
        this.MyString = reader.ReadString();
    }
}
```

If a `NetworkBehaviour` has a base class that also has serialization functions, the base class functions should also be called.

Note that the `UpdateVar` packets created for GameObject state updates may be aggregated in buffers before being sent to the client, so a single transport layer packet may contain updates for multiple GameObjects.

Network visibility

Leave feedback

Multiplayer games use the concept of **network visibility** to determine which players can see which **GameObjects** at any given time during gameplay. In a game that has a moving viewpoint and moving **GameObjects**, it's common that players cannot see everything that is happening in the game at once.

If a particular player, at a certain point in time during gameplay, cannot see most of the other players, non-player characters, or other moving or interactive things in your game, there is usually no need for the host to send information about those things to the player's client.

This can benefit your game in two ways:

It reduces the **amount of data** sent across the network between players. This can help improve the responsiveness of your game, and reduce bandwidth use. The bigger and more complex your multiplayer game, the more important this issue is.

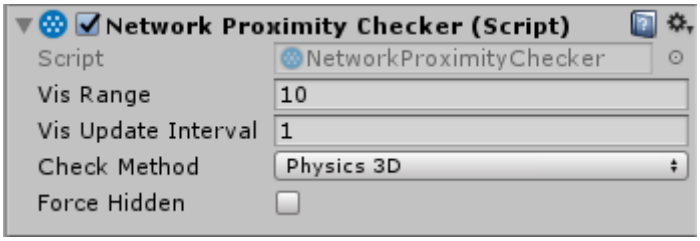
It also helps** prevent hacking**. Since a player client does not have information about things that can't be seen, a hack on that player's computer cannot reveal the information.

The idea of "visibility" in the context of **networking** doesn't necessarily relate to whether GameObjects are directly visible on-screen. Instead, it relates to whether data should or shouldn't be sent about the GameObject in question to a particular client. Put simply, if a client can't 'see' an GameObject, it does not need to be sent information about that GameObject across the network. Ideally you want to limit the amount of data you are sending across the network to only what is necessary, because sending large amounts of unnecessary data across the network can cause network performance problems.

However, it can be also be resource intensive or complex to determine accurately whether a GameObject truly visible to a given player, so it's often a good idea to use a more simple calculation for the purposes of determining whether a player should be sent networked data about it - i.e. whether it is 'Network Visible'. The balance you want to achieve when considering this is between the cost of the complexity of the calculation for determining the visibility, and the cost of sending more information than necessary over the network. A very simple way to calculate this is a distance (proximity) check, and Unity provides a built-in **component** for this purpose.

Network Proximity Checker component

Unity's Network Proximity Checker component is simplest way to implement network visibility for players. It works in conjunction with the physics system to determine whether GameObjects are close enough (that is, "visible" for the purposes of sending network messages in your multiplayer game).



The Network Proximity Checker component

To use this component, add it to the **Prefab** of the networked GameObject for which you want to limit network visibility.

The Network Proximity Checker has two configurable visibility parameters:

Vis Range controls the distance threshold within which the network should consider a GameObject visible to a player.

Vis Update Interval controls how often the distance test is performed. The value is the delay in seconds between checks. This allows you to optimise the check to perform as few tests as possible. The lower the number, the more frequently the updates occur. For slow-moving GameObjects you can set this interval to higher values. For fast-moving GameObjects, you should set it to lower values.

You must attach a **Collider** component to any GameObjects you want to use with the Network Proximity Checker.

Network visibility on remote clients

When a player on a remote client joins a networked game, only GameObjects that are network-visible to the player will be spawned on that remote client. This means that even if the player enters a large world with many networked GameObjects, the game can start quickly because it does not need to spawn every GameObject that exists in the world. Note that this applies to networked GameObjects in your **Scene**, but does not affect the loading of **Assets**. Unity still takes time to load the **Assets** for registered Prefabs and **Scene** GameObjects.

When a player moves within the world, the set of network-visible GameObjects changes. The player's client is told about these changes as they happen. The ObjectHide** **message is sent to clients when a GameObject becomes no longer network-visible. By default, Unity destroys the GameObject when it receives this message. When a GameObject becomes visible, the client receives an ObjectSpawn message, as if Unity has spawned the GameObject for the first time. By default, the GameObject is instantiated like any other spawned GameObject.

Network visibility on the host

The host shares the same Scene as the server, because it acts as both the server and the client to the player hosting the game. For this reason, it cannot destroy GameObjects that are not visible to the local player.

Instead, there is the virtual methodOnSetLocalVisibility** **on the NetworkBehaviour class that is invoked. This method is invoked on all `NetworkBehaviour` **scripts** on GameObjects that change visibility state on the host.

The default implementation of `OnSetLocalVisibility` disables or enables all **Renderer** components on the GameObject. If you want to customize this implementation, you can override the method in your script, and provide a new behaviour for how the host (and therefore the local client) should respond when a GameObject becomes network-visible or invisible (such as disabling HUD elements or renderers).

Customizing network visibility

Leave feedback

The built-in Network Proximity Checker **component** is the built-in default **component** for determining a **GameObject**'s network visibility. However, this only provides you with a distance-based check. Sometimes you might want to use other kinds of visibility check, such as grid-based rules, line-of-sight tests, navigation path tests, or any other type of test that suits your game.

To do this, you can implement your own custom equivalent of the Network Proximity Checker. To do that, you need to understand how the Network Proximity Checker works. See documentation on the in-editor Network Proximity Checker component, and the NetworkProximityChecker API.

The Network Proximity Checker is implemented using the public **visibility interface** of Unity's Multiplayer HLAPI. Using this same interface, you can implement any kind of visibility rules you desire. Each NetworkIdentity keeps track of the set of players that it is visible to. The players that a NetworkIdentity GameObject is visible to are called the "observers" of the NetworkIdentity.

The Network Proximity Checker calls the RebuildObservers** **method on the Network Identity component at a fixed interval (set using the "**Vis Update Interval**" value in the inspector), so that the set of network-visible GameObjects for each player is updated as they move around.

On the `NetworkBehaviour`** **class (which your networked scripts inherit from), there are some virtual functions for determining visibility. These are:

OnCheckObserver - This method is called on the server, on each networked GameObject when a new player enters the game. If it returns true, that player is added to the object's observers. The `NetworkProximityChecker` method does a simple distance check in its implementation of this function, and uses `Physics.OverlapSphere()` to find the players that are within the visibility distance for this object.

OnRebuildObservers - This method is called on the server when `RebuildObservers` is invoked. This method expects the set of observers to be populated with the players that can see the object. The NetworkServer then handles sending `ObjectHide` and `ObjectSpawn` messages based on the differences between the old and new visibility sets.

You can check whether any given networked GameObject is a player by checking if its `NetworkIdentity` has a valid connectionToClient. For example:

```
var hits = Physics.OverlapSphere(transform.position, visRange);
foreach (var hit in hits)
{
    // (if a GameObject has a connectionToClient, it is a player)
    var uv = hit.GetComponent<NetworkIdentity>();
    if (uv != null && uv.connectionToClient != null)
    {
        observers.Add(uv.connectionToClient);
    }
}
```

Scene GameObjects

Leave feedback

There are two types of networked GameObjects in Unity's multiplayer system:

Those that are created dynamically at runtime

Those that are saved as part of a Scene

GameObjects that are created dynamically at runtime use the multiplayer Spawning system, and the **prefabs** they are instantiated from must be registered in the **Network Manager**'s list of networked **GameObject prefabs**.

However, networked GameObjects that you save as part of a **Scene** (and therefore already exist in the Scene when it is loaded) are handled differently. These GameObjects are loaded as part of the Scene on both the client and server, and exist at runtime before any spawn messages are sent by the multiplayer system.

When the Scene is loaded, all networked GameObjects in the Scene are disabled** **on both the client and the server. Then, when the Scene is fully loaded, the Network Manager automatically processes the Scene's networked GameObjects, registering them all (and therefore causing them to be synchronized across clients), and enabling them, as if they were spawned at runtime.

Saving networked GameObjects in your Scene (rather than dynamically spawning them after the scene has loaded) has some benefits:

They are loaded with the level, so there will be no pause at runtime.
They can have specific modifications that differ from prefabs
Other GameObject instances in the Scene can reference them, which can avoid you having to use code to finding the GameObjects and make references to them up at runtime.
When the Network Manager spawns the networked Scene GameObjects, those GameObjects behave like dynamically spawned GameObjects. Unity sends them updates and ClientRPC calls.

If a Scene GameObject is destroyed on the server before a client joins the game, then it is never enabled on new clients that join.

When a client connects, the client is sent an ObjectSpawnScene spawn message for each of the Scene GameObjects that exist on the server, that are visible to that client. This message causes the GameObject on the client to be enabled, and has the latest state of that GameObject from the server in it. This means that only GameObjects that are visible to the client, and not destroyed on the server, are activated on the client. Like regular non-Scene GameObjects, these Scene GameObjects are started with the latest state when the client joins the game.

Actions and communication

Leave feedback

When you are making a multiplayer game, In addition to synchronizing the properties of networked GameObjects, you are likely to need to send, receive, and react to other pieces of information - such as when the match starts, when a player joins or leaves the match, or other information specific to your type of game, for example a notification to all players that a flag has been captured in a “capture-the-flag” style game.

Within the Unity **networking** High-Level API there are three main ways to communicate this type of information.

Remote actions

Remote actions allow you to call a method in your script across the network. You can make the server call methods on all clients or individual clients specifically. You can also make clients call methods on the server. Using remote actions, you can pass data as parameters to your methods in a very similar way to how you call methods in local (non-multiplayer) projects.

Networking callbacks

Networking callbacks allow you to hook into built-in Unity events which occur during the course of the game, such as when players join or leave, when **GameObjects** are created or destroyed, or when a new **Scene** is loaded. There are two types of networking callbacks that you can implement:

Network manager callbacks, for callbacks relating to the **network manager** itself (such as when clients connect or disconnect)

Network behaviour callbacks, for callbacks relating to individual networked GameObjects (such as when its Start function is called, or what this particular GameObject should do if a new player joins the game)

Network messages

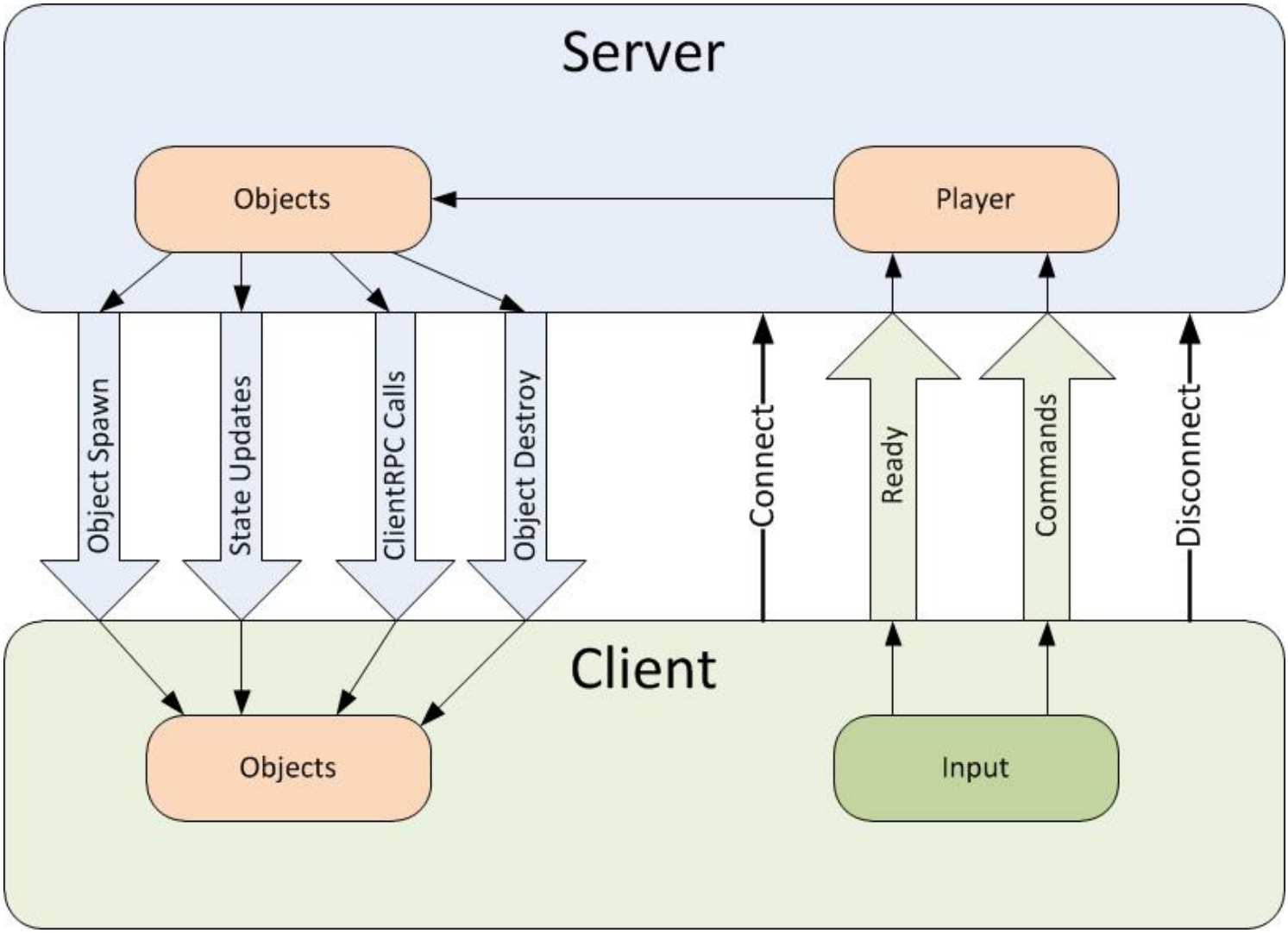
Network messages are a “lower level” approach to sending messages (although they are still classed as part of the networking “High level API”). They allow you to send data directly between clients and the server using scripting. You can send basic types of data (int, string, etc) as well as most common Unity types (such as Vector3). Since you implement this yourself, these messages are not associated directly with any particular GameObjects or Unity events - it is up to you do decide their purpose and implement them!

Remote Actions

Leave feedback

The network system has ways to perform actions across the network. These type of actions are sometimes called Remote Procedure Calls. There are two types of RPCs in the network system, Commands - which are called from the client and run on the server; and ClientRpc calls - which are called on the server and run on clients.

The diagram below shows the directions that remote actions take:



Commands

Commands are sent from player objects on the client to player objects on the server. For security, Commands can only be sent from YOUR **player object**, so you cannot control the objects of other players. To make a function into a command, add the [Command] custom attribute to it, and add the “Cmd” prefix. This function will now be run on the server when it is called on the client. Any arguments will automatically be passed to the server with the command.

Commands functions must have the prefix “Cmd”. This is a hint when reading code that calls the command - this function is special and is not invoked locally like a normal function.

```
class Player : NetworkBehaviour
{
```

```
    public GameObject bulletPrefab;

    [Command]
    void CmdDoFire(float lifeTime)
    {
        GameObject bullet = (GameObject)Instantiate(
            bulletPrefab,
            transform.position + transform.right,
            Quaternion.identity);

        var bullet2D = bullet.GetComponent<Rigidbody2D>();
        bullet2D.velocity = transform.right * bulletSpeed;
        Destroy(bullet, lifeTime);

        NetworkServer.Spawn(bullet);
    }

    void Update()
    {
        if (!isLocalPlayer)
            return;

        if (Input.GetKeyDown(KeyCode.Space))
        {
            CmdDoFire(3.0f);
        }

    }
}
```

Be careful of sending commands from the client every frame! This can cause a lot of network traffic.

By default, Commands are sent on channel zero - the default reliable channel. So by default all commands are reliably sent to the server. This can be customized with the “Channel” parameter of the [Command] custom attribute. This parameter should be an integer, representing the channel number.

Channel 1 is also set up by default to be an unreliable channel, so to use this, use the value 1 for the parameter in the Command attribute, like this:

```
    [Command(channel=1)]
```

Starting with Unity release 5.2 it is possible to send commands from non-player objects that have client authority. These objects must have been spawned with NetworkServer.SpawnWithClientAuthority or have authority set with NetworkIdentity.AssignClientAuthority. Commands sent from these object are run on the server instance of the object, not on the associated player object for the client.

ClientRpc Calls

ClientRpc calls are sent from objects on the server to objects on clients. They can be sent from any server object with a **NetworkIdentity** that has been spawned. Since the server has authority, then there no security issues with server objects being able to send these calls. To make a function into a ClientRpc call, add the [ClientRpc] custom attribute to it, and add the "Rpc" prefix. This function will now be run on clients when it is called on the server. Any arguments will automatically be passed to the clients with the ClientRpc call..

ClientRpc functions must have the prefix "Rpc". This is a hint when reading code that calls the method - this function is special and is not invoked locally like a normal function.

```
class Player : NetworkBehaviour
{

    [SyncVar]
    int health;

    [ClientRpc]
    void RpcDamage(int amount)
    {
        Debug.Log("Took damage:" + amount);
    }

    public void TakeDamage(int amount)
    {
        if (!isServer)
            return;

        health -= amount;
        RpcDamage(amount);
    }
}
```

When running a game as a host with a LocalClient, ClientRpc calls will be invoked on the LocalClient - even though it is in the same process as the server. So the behaviour of LocalClients and RemoteClients is the same for ClientRpc calls.

Arguments to Remote Actions

The arguments passed to commands and ClientRpc calls are serialized and sent over the network. These arguments can be:

basic types (byte, int, float, string, UInt64, etc)
arrays of basic types
structs containing allowable types
built-in unity math types (Vector3, Quaternion, etc)
NetworkIdentity
NetworkInstanceId
NetworkHash128
GameObject with a NetworkIdentity **component** attached

Arguments to remote actions cannot be subcomponents of GameObjects, such as script instances or Transforms. They cannot be other types that cannot be serialized across the network.

Network Manager callbacks

Leave feedback

There are a number of events that can occur over the course of the normal operation of a multiplayer game, such as the host starting up, a player joining, or a player leaving. Each of these possible events has an associated **callback** that you can implement in your own code to take action when the event occurs.

To do this for the **Network Manager**, you need to create your own script which inherits from NetworkManager. You can then override the virtual methods on NetworkManager with your own implementation of what should happen when the given event occurs.

This page lists all the virtual methods (the callbacks) that you can implement on the Network Manager, and when they occur. The callbacks that occur, and the order they occur, vary slightly depending on whether your game is running in LAN mode or Internet (matchmaker) mode, so each mode's callbacks are listed separately below.

LAN Callbacks

These are the callbacks that occur when the game is running on a Local Area Connection (LAN). A game can be running in one of three modes, **host**, **client**, or **server-only**. The callbacks for each mode are listed below:

LAN callbacks in host mode:

When the host is started:

`Start()` function is called

`OnStartHost`

`OnStartServer`

`OnServerConnect`

`OnStartClient`

`OnClientConnect`

`OnServerSceneChanged`

`OnServerReady`

`OnServerAddPlayer`

`OnClientSceneChanged`

When a client connects:

`OnServerConnect`

`OnServerReady`

`OnServerAddPlayer`

When a client disconnects:

`OnServerDisconnect`

When the host is stopped:

`OnStopHost`

`OnStopServer`

`OnStopClient`

LAN callbacks in client mode

When the client starts:

`Start()` function is called

`OnStartClient`

`OnClientConnect`

`OnClientSceneChanged`

When the client stops:

`OnStopClient`

`OnClientDisconnect`

LAN callbacks in server mode

When the server starts:

`Start()` function is called

`OnStartServer`

`OnServerSceneChanged`

When a client connects:

`OnServerConnect`

`OnServerReady`

`OnServerAddPlayer`

When a client disconnects:

`OnServerDisconnect`

When the server stops:

`OnStopServer`

MatchMaker connection callbacks

These are the callbacks which occur when the game is running in Internet mode (i.e. when you are using the MatchMaker service to find and connect to other players. In this mode, a game can be running in one of two modes, **host**, or **client**. The callbacks for each mode are listed below:

MatchMaker callbacks in host mode

When the host starts:

`Start()` function is called

`OnStartHost`

`OnStartServer`

`OnServerConnect`

`OnStartClient`

`OnMatchCreate`

`OnClientConnect`

`OnServerSceneChanged`

`OnServerReady`

`OnServerAddPlayer`

`OnClientSceneChanged`

When a client connects:

`OnServerConnect`

`OnServerReady`

`OnServerAddPlayer`

When a client disconnects:

`OnServerDisconnect`

MatchMaker callbacks in client mode

When receiving a list of online game instances:

`Start()` function is called

`OnMatchList`

When joining a match:

`OnStartClient`

`OnMatchJoined`

`OnClientConnect`

`OnClientSceneChanged`

When the host stops:

`OnStopClient`

`OnClientDisconnect`

NetworkBehaviour callbacks

Leave feedback

Like the Network Manager callbacks, there are a number of events relating to network behaviours that can occur over the course of a normal multiplayer game. These include events such as the host starting up, a player joining, or a player leaving. Each of these possible events has an associated **callback** that you can implement in your own code to take action when the event occurs.

When you create a script which **inherits** from NetworkBehaviour, you can write your own implementation of what should happen when these events occur. To do this, you override the virtual methods on the `NetworkBehaviour` class with your own implementation of what should happen when the given event occurs.

This page lists all the virtual methods (the callbacks) that you can implement on Network Behaviour, and when they occur. A game can run in one of three modes, **host**, **client**, or **server-only**. The callbacks for each mode are listed below:

Callbacks in server mode

When a client connects:

`OnStartServer`

`OnRebuildObservers`

`Start()` function is called

Callbacks in client mode

When a client connects:

`OnStartClient`

`OnStartLocalPlayer`

`OnStartAuthority`

`Start()` function is called

Callbacks in host mode

These are only called on the **Player GameObjects** when a client connects:

`OnStartServer`

`OnStartClient`

`OnRebuildObservers`

`OnStartAuthority`

`OnStartLocalPlayer`

`Start()` function is called

`OnSetLocalVisibility`

On any remaining clients, when a client disconnects:

`OnNetworkDestroy`

Network Messages

Leave feedback

In addition to “high-level” Commands and RPC calls, you can also send raw network messages.

There is a class called MessageBase that you can extend to make serializable network message classes. This class has Serialize and Deserialize functions that take writer and reader objects. You can implement these functions yourself, or you can rely on code-generated implementations that are automatically created by the **networking** system. The base class looks like this:

```
public abstract class MessageBase
{
    // De-serialize the contents of the reader into this message
    public virtual void Deserialize(NetworkReader reader) {}

    // Serialize the contents of this message into the writer
    public virtual void Serialize(NetworkWriter writer) {}
}
```

Message classes can contain members that are basic types, structs, and arrays, including most of the common Unity Engine types (such as Vector3). They cannot contain members that are complex classes or generic containers. Remember that if you want to rely on the code-generated implementations, you must make sure your types are publicly visible.

There are built-in message classes for common types of network messages:

EmptyMessage

StringMessage

IntegerMessage

To send a message, use the `Send()` method on the NetworkClient, NetworkServer, and NetworkConnection classes which work the same way. It takes a message ID, and a message object that is derived from MessageBase. The code below demonstrates how to send and handle a message using one of the built-in message classes:

```
using UnityEngine;
using UnityEngine.Networking;
using UnityEngine.Networking.NetworkSystem;

public class Begin : NetworkBehaviour
{
    const short MyBeginMsg = 1002;

    NetworkClient m_client;

    public void SendReadyToBeginMessage(int myId)
    {
        var msg = new IntegerMessage(myId);
        m_client.Send(MyBeginMsg, msg);
    }

    public void Init(NetworkClient client)
    {
```

```
        m_client = client;
        NetworkServer.RegisterHandler(MyBeginMsg, OnServerReadyToBeginMessage);
    }

    void OnServerReadyToBeginMessage(NetworkMessage netMsg)
    {
        var beginMessage = netMsg.ReadMessage<IntegerMessage>();
        Debug.Log("received OnServerReadyToBeginMessage " + beginMessage.value);
    }
}
```

To declare a custom network message class and use it:

```
using UnityEngine;
using UnityEngine.Networking;

public class Scores : MonoBehaviour
{
    NetworkClient myClient;

    public class MyMsgType {
        public static short Score = MsgType.Highest + 1;
    };

    public class ScoreMessage : MessageBase
    {
        public int score;
        public Vector3 scorePos;
        public int lives;
    }

    public void SendScore(int score, Vector3 scorePos, int lives)
    {
        ScoreMessage msg = new ScoreMessage();
        msg.score = score;
        msg.scorePos = scorePos;
        msg.lives = lives;

        NetworkServer.SendToAll(MyMsgType.Score, msg);
    }

    // Create a client and connect to the server port
    public void SetupClient()
    {
        myClient = new NetworkClient();
        myClient.RegisterHandler(MsgType.Connect, OnConnected);
        myClient.RegisterHandler(MyMsgType.Score, OnScore);
        myClient.Connect("127.0.0.1", 4444);
    }
```

```
    public void OnScore(NetworkMessage netMsg)
    {
        ScoreMessage msg = netMsg.ReadMessage<ScoreMessage>();
        Debug.Log("OnScoreMessage " + msg.score);
    }

    public void OnConnected(NetworkMessage netMsg)
    {
        Debug.Log("Connected to server");
    }
}
```

Note that there is no serialization code for the `ScoreMessage` class in this source code example. The body of the serialization functions is automatically generated for this class by Unity.

ErrorMessage class

There is also an ErrorMessage class that is derived from `MessageBase`. This class is passed to error callbacks on clients and servers.

The errorCode in the ErrorMessage class corresponds to the Networking.NetworkError enumeration.

```
class MyClient
{
    NetworkClient client;

    void Start()
    {
        client = new NetworkClient();
        client.RegisterHandler(MsgType.Error, OnError);
    }

    void OnError(NetworkMessage netMsg)
    {
        var errorMsg = netMsg.ReadMessage<ErrorMessage>();
        Debug.Log("Error:" + errorMsg.errorCode);
    }
}
```

Dealing with clients and servers

Leave feedback

When you are making your multipayer game, you will need to implement a way for players to find each other, join existing matches or create new ones. You will also need to decide how to deal with common network problems, such as what happens if the person hosting the game quits.

This section provides information on how to build these important parts of your game, including:

Host migration, for when the player hosting a peer-hosted game quits.

Network discovery, to help your players connect to each other on a LAN

Multiplayer lobby, to help your players create or join matches across the internet

Custom network client and server code - when you have custom requirements and want to write your own connection code rather than using Unity's Network Manager.

Network clients and servers

Leave feedback

Many multiplayer games can use the Network Manager to manage connections, but you can also use the lower-level NetworkServer and NetworkClient classes directly.

When using the High-Level API, every game must have a host server to connect to. Each participant in a multiplayer game can be a client, a dedicated server, or a combination of server and client at the same time. This combination role is the common case of a multiplayer game with no dedicated server.

For multiplayer games with no dedicated server, one of the players running the game acts as the server for that game. That player's instance of the game runs a "local client" instead of a normal remote client. The local client uses the same Unity **Scenes** and **GameObjects** as the server, and communicates internally using message queues instead of sending messages across the network. To HLAPI code and systems, the local client is just another client, so almost all user code is the same, whether a client is local or remote. This makes it easy to make a game that works in both multiplayer and standalone mode with the same code.

A common pattern for multiplayer games is to have a GameObject that manages the network state of the game. Below is the start of a NetworkManager script. This script would be attached to a GameObject that is in the start-up Scene of the game. It has a simple **UI** and keyboard handling functions that allow the game to be started in different network modes. Before you release your game you should create a more visually appealing menu, with options such as "Start single player game" and "Start multiplayer game".

```
using UnityEngine;
using UnityEngine.Networking;

public class MyNetworkManager : MonoBehaviour {
    public bool isAtStartup = true;
    NetworkClient myClient;
    void Update ()
    {
        if (isAtStartup)
        {
            if (Input.GetKeyDown(KeyCode.S))
            {
                SetupServer();
            }
            if (Input.GetKeyDown(KeyCode.C))
            {
                SetupClient();
            }
            if (Input.GetKeyDown(KeyCode.B))
            {
                SetupServer();
                SetupLocalClient();
            }
```

```
        }
    }
    void OnGUI()
    {
        if (isAtStartup)
        {
            GUI.Label(new Rect(2, 10, 150, 100), "Press S for server");
            GUI.Label(new Rect(2, 30, 150, 100), "Press B for both");
            GUI.Label(new Rect(2, 50, 150, 100), "Press C for client");
        }
    }
}
```

This basic code calls setup functions to get things going. Below are the simple setup functions for each of the scenarios. These functions create a server, or the right kind of client for each scenario. Note that the remote client assumes the server is on the same machine (127.0.0.1). For a finished game this would be an internet address, or something supplied by the Matchmaking system.

```
// Create a server and listen on a port
    public void SetupServer()
    {
        NetworkServer.Listen(4444);
        isAtStartup = false;
    }

    // Create a client and connect to the server port
    public void SetupClient()
    {
        myClient = new NetworkClient();
        myClient.RegisterHandler(MsgType.Connect, OnConnected);
        myClient.Connect("127.0.0.1", 4444);
        isAtStartup = false;
    }

    // Create a local client and connect to the local server
    public void SetupLocalClient()
    {
        myClient = ClientScene.ConnectLocalServer();
        myClient.RegisterHandler(MsgType.Connect, OnConnected);
        isAtStartup = false;
    }
```

The clients in this code register a callback function for the connection event MsgType.Connect. This is a built-in message of the HLAPI that the script invokes when a client connects to a server. In this case, the code for the handler on the client is:

```
// client function
    public void OnConnected(NetworkMessage netMsg)
    {
        Debug.Log("Connected to server");
    }
```

This is enough to get a multiplayer application up and running. With this script you can then send network messages using NetworkClient.Send and NetworkServer.SendToAll. Note that sending messages is a low level way of interacting with the system.

Host Migration

Leave feedback

In a multiplayer network game without a dedicated server, one of the game instances acts as the host - the center of authority for the game. This is a player whose game is acting as a server and a “local client”, while the other players each run a “remote client”. See documentation on network system concepts for more information.

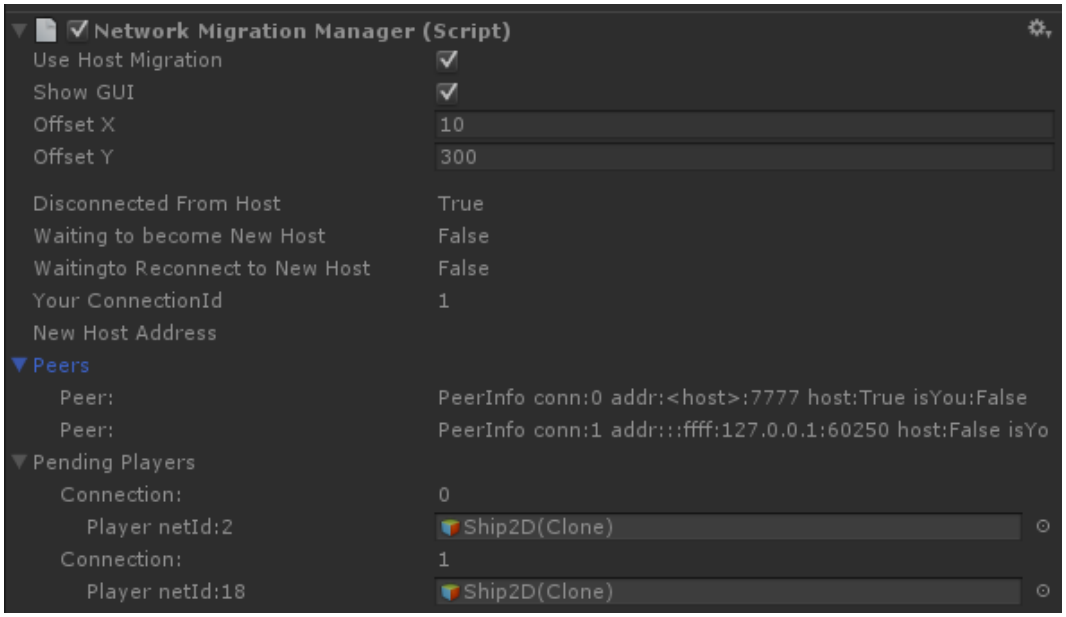
If the host disconnects from the game, gameplay cannot continue. Common reasons for a host to disconnect include the host player leaving, the host process crashing, the host’s machine shutting down, or the host losing network connection.

The **host migration** feature allows one of the remote clients to become the new host, so that the multiplayer game can continue.

How it works

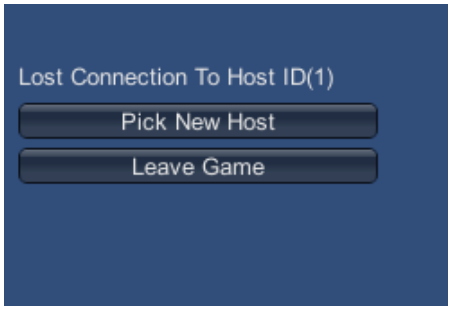
During a multiplayer game with host migration enabled,Unity distributes the addresses of all peers (players, including the host and all clients) to all other peers in the game. When the host disconnects, one peer becomes the new host. The other peers then connect to the new host, and the game continues.

The Network Migration Manager **component** uses the Unity **Networking** HLAPI. It allows the game to continue with a new host after the original host disconnects. The screenshot below shows the migration state displayed in the Network Migration Manager, in the **Inspector** window.



The Network Migration Manager component

The Network Migration Manager provides a basic user interface, similar to the Network Manager HUD. This user interface is for testing and prototyping during game development; before you release your game you should implement a custom user interface for host migration, and custom logic for actions like choosing the new host automatically without requiring input from the user.



The Network Migration Manager prototyping HUD

Even though the migration may have occurred because the old host lost connection or quit the game, it is possible for the old host of the game to rejoin the game as a client on the new host.

During a host migration, Unity maintains the state of SyncVars and SyncLists on all networked **GameObjects** in the **Scene**. This also applies to custom serialized data for **GameObjects**.

Unity disables all of the player GameObjects in the game when the host disconnects. Then, when the other clients rejoin the new game on the new host, the corresponding players Unit re-enables those clients on the new host, and respawns them on the other clients. This ensures that Unity does not lose player state data during a host migration.

NOTE: During a host migration, Unity only preserves data that is available to clients. If data is only on the server, then it is not available to the client that becomes the new host. Data on the host is only available after a host migration if it is in SyncVars or SyncLists.

When the client becomes a new host, Unity invokes the callback function OnStartServer for all networked GameObjects. On the new host, the Network Migration Manager uses the function BecomeNewHost to construct a networked server Scene from the state in the current ClientScene.

In a game with host migration enabled, peers are identified by their connectionId on the server. When a client reconnects to the new host of a game, Unity passes this connectionId to the new host so that it can match this client with the client that was connected to the old host. This Id is on the `ClientScene` as the reconnectId.

Non-Player GameObjects

Non-player GameObjects with client authority are also handled by host migration. Unity disables and re-enables client-owned non-player GameObjects in the same way it disables and re-enables player GameObjects.

Identifying Peers

Before the host disconnects, all the peers are connected to the host. They each have a unique connectionId on the host - this is called the oldConnectionId in the context of host migration.

When the Network Migration Manager chooses a new host, and the peers reconnect to it, they supply their `oldConnectionId` to identify which peer they are. This allows the new host to match this reconnecting client to the corresponding player GameObject.

The old host uses a special `oldConnectionId` of zero to reconnect - because it did not have a connection to the old host, it WAS the old host. There is a constant ClientScene.ReconnectIdHost for this.

When you use the Network Migration Manager's built-in user interface, the Network Migration Manager sets the `oldConnectionId` automatically. To set it manually, use NetworkMigrationManager.Reset or ClientScene.SetReconnectId.

Host Migration Flow

MachineA hosts Game1, a game with host migration enabled

MachineB starts a client and joins Game1

MachineB is told about peers (MachineA–0, and self (MachineB)–1)
MachineC starts a client and joins Game1

MachineC is told about peers (MachineA–0, MachineB–1, and self (MachineC)–2)
MachineA drops the connection on Game 1, so the host disconnects

MachineB disconnects from host

MachineB callback function is invoked on MigrationManager on client

MachineB player GameObjects for all players are disabled

MachineB stays in online Scene

MachineB uses utility function to pick the new host, picks self

MachineB calls BecomeNewHost()

MachineB start listening

MachineB player GameObject for self is reactivated

MachineB The player for MachineB is now back in the game with all its old state

MachineC gets disconnect from host

MachineC callback function is invoked on MigrationManager on client

MachineC player GameObjects for all players are disabled

MachineC stay in online Scene

MachineC uses utility function to pick new host, picks MachineB

MachineC reconnects to new host

MachineB receives connection from MachineC

MachineC send reconnect message with oldConnectionId (instead of AddPlayer message)

callback function is invoked on MigrationManager on server

MachineB uses oldConnectionId to find the disabled player GameObject for that player and re-adds it with ReconnectPlayerForConnection()

player GameObject is re-spawned for MachineC

The player for MachineC is now back in the game with all its old state

MachineA recovers (the old host)

MachineA uses utility function to pick the new host, picks MachineB

MachineA “reconnects” to MachineB

MachineB receives connection from MachineA

MachineA send reconnect message with oldConnectionId of zero

callback function is invoked on MigrationManager on server (MachineB)

MachineB uses oldConnectionId to find the disabled player GameObject for that player and re-adds it with ReconnectPlayerForConnection()

player GameObject is re-spawned for MachineA

The player for MachineA is now back in the game with all its old state

Callback Functions

Callback functions on the NetworkHostMigrationManager:

```
// called on client after the connection to host is lost. controls whether to switch Scene
protected virtual void OnClientDisconnectedFromHost(
    NetworkConnection conn,
    out SceneChangeOption sceneChange)

// called on host after the host is lost. host MUST change Scenes
protected virtual void OnServerHostShutdown()

// called on new host (server) when a client from the old host re-connects a player
protected virtual void OnServerReconnectPlayer(
    NetworkConnection newConnection,
    GameObject oldPlayer,
    int oldConnectionId,
    short playerControllerId)

// called on new host (server) when a client from the old host re-connects a player
protected virtual void OnServerReconnectPlayer(
    NetworkConnection newConnection,
    GameObject oldPlayer,
    int oldConnectionId,
    short playerControllerId,
    NetworkReader extraMessageReader)

// called on new host (server) when a client from the old host re-connects a non-player Ga
protected virtual void OnServerReconnectObject(
    NetworkConnection newConnection,
    GameObject oldObject,
    int oldConnectionId)

// called on both host and client when the set of peers is updated
protected virtual void OnPeersUpdated(
    PeerListMessage peers)

// utility function called by the default UI on client after connection to host was lost,
public virtual bool FindNewHost(
    out NetworkSystem.PeerInfoMessage newHostInfo,
    out bool youAreNewHost)

// called when the authority of a non-player GameObject changes
protected virtual void OnAuthorityUpdated(
    GameObject go,
    int connectionId,
    bool authorityState)
```

Constraints

For host migration to work properly, you need to go to the GameObject's **Network Manager** component and enable **Auto Create Player**. Data that is only present on the server (the host) is lost when the host disconnects. For games to be able to perform host migration correctly, important data must be distributed to the clients, not held secretly on the server.

This works for direct connection games. Additional work is required for this to function with the matchmaker and relay server.

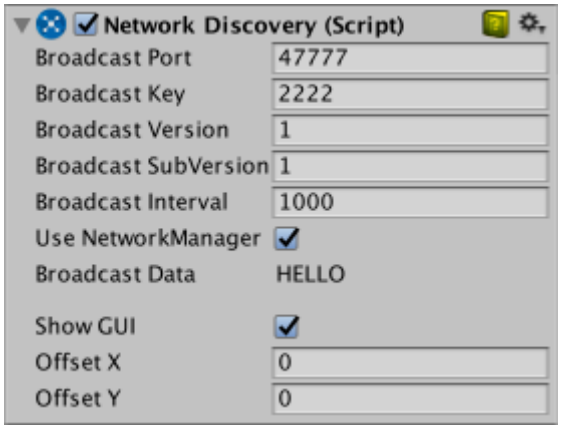
Network Discovery

Leave feedback

The Network Discovery component allows Unity multiplayer games to find each other on a local area network (a LAN). This means your players don't have to find out the IP address of the host to connect to a game on a LAN. Network Discovery doesn't work over the internet, only on local networks. For internet-based games, see the MatchMaker service.

The Network Discovery **component** can broadcast its presence, listen for broadcasts from other Network Discovery components, and optionally join matching games using the Network Manager. The Network Discovery component uses the UDP broadcast feature of the network transport layer.

To use local network discovery, create an empty **GameObject** in the **Scene**, and add the Network Discovery component to it.



NetworkDiscovery Component

Like the Network Manager HUD, this component has a default GUI that shows in the Game view for controlling it, intended for temporary developmental work, with the assumption that you will create your own replacement for it before finishing your game. Note that you also need a Network Manager component in the Scene to be able to join a game through the Network Discovery GUI. When the game starts, click the **Initialize Broadcast** button in the Network Discovery GUI (in the Game view) to send a broadcast and begin discovery of other games on the local network.

The Network Discovery component can run in server mode (activated by clicking the "Start Broadcasting" button in the GUI), or client mode (activated by clicking the 'Listen for Broadcast' button in the GUI).

When in **server mode**, the Network Discovery component sends broadcast messages over the network on the port specified in the **inspector**. These messages contain the **Broadcast Key** and **Broadcast Version** of the game. You can set these to any value you like, their purpose is to identify this particular version and release of your game to avoid conflicts - such as your game trying to join a game of a different type. You should change the **Broadcast Key** value when releasing a new build of your game that should not be able to connect to older versions of your game. The component should be run in server mode if a game is being hosted on that machine. Without the default GUI, you need to call the StartAsServer() function to make the component run in server mode.

When in **client mode**, the component listens for broadcast messages on the specified port. When a message is received, and the **Broadcast Key** in the message matches the **Broadcast Key** in the Network Discovery component, this means that a game is available to join on the local network. Without the default GUI, you need to call the StartAsClient() function to make the component run in client mode.

When using the default GUI and listening for broadcasts in client mode, if a game is discovered on the local network, a button appears which allows the user of that client to join the game. The button is labeled “Game at:” followed by the host’s IP address.

There is a virtual function on the Network Discovery component that can be implemented to be notified when broadcast messages are received.

```
public class MyNetworkDiscovery: NetworkDiscovery {
    public override void OnReceivedBroadcast(string fromAddress, string data)
    {
        Debug.Log("Received broadcast from: " + fromAddress+ " with the data: "
    }
}
```

For more information, see the Scripting API Reference documentation on NetworkDiscovery. Note that you cannot have a Network Discovery server and client running in the same process at the same time.

Multiplayer Lobby

Leave feedback

Most multiplayer games have a “lobby”; a **Scene** in the game for players to join before playing the actual game. In the lobby, players can pick options and set themselves as ready for the game to start.

Unity provides the Network Lobby Manager component as a way for you to implement a Lobby for your game easily.

The Network Lobby Manager **component** provides a lobby for Unity Multiplayer games. It includes the following features:

A simple built-in user interface for interacting with the lobby
Limits the number of players that can join
Supports multiple players per client, with a limit on number of players per client
Prevents players from joining games that are in-progress
Supports a “ready” state for clients, so a game starts when all players are ready
Configuration data for each player
Players re-join the lobby when the game finishes
Virtual functions that allow custom logic for lobby events

Below are the Network Lobby Manager virtual methods. See API Reference documentation on the NetworkLobbyManager class for more details. There is a separate list for methods that are called on the client and on the server. You can write your own implementations for these methods to take action when any of these events occur.

`NetworkLobbyManager` virtual methods called on the **server**:

OnLobbyStartHost
OnLobbyStopHost
OnLobbyStartServer
OnLobbyServerConnect
OnLobbyServerDisconnect
OnLobbyServerSceneChanged
OnLobbyServerCreateLobbyPlayer
OnLobbyServerCreateGamePlayer
OnLobbyServerPlayerRemoved
OnLobbyServerSceneLoadedForPlayer
OnLobbyServerPlayersReady

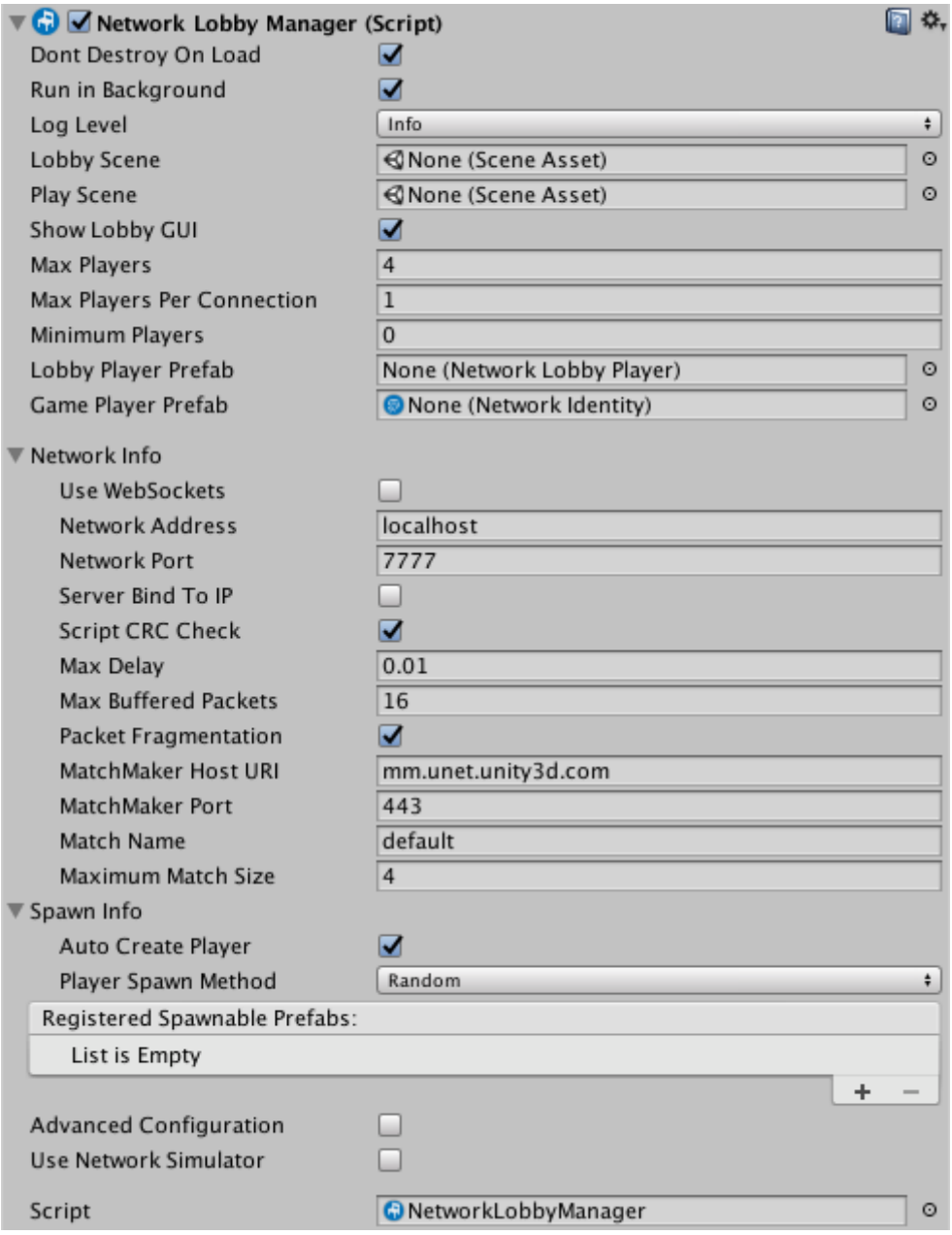
`NetworkLobbyManager` virtual methods called on the **client**:

OnLobbyClientEnter
OnLobbyClientExit
OnLobbyClientConnect
OnLobbyClientDisconnect
OnLobbyStartClient
OnLobbyStopClient
OnLobbyClientSceneChanged
OnLobbyClientAddPlayerFailed

All of the above server and client methods have empty default implementations, except for `OnLobbyServerPlayersReady`**, which calls ServerChangeScene with the PlayScene (the scene assigned to the **Play Scene** field in the Lobby Manager inspector.

Lobby Player GameObjects

There are two kinds of player Prefabs for the Lobby Manager: the **Lobby Player Prefab** and the** Game Player **Prefab**. There is a field for each in the Network Lobby Manager component.



The Network Lobby Manager component

Lobby Player Prefab

The Prefab that you assign to the **Lobby Player Prefab** slot must have a Network Lobby Player component attached. Each client that joins the Lobby gets a new Lobby player **GameObject**, created from the **Lobby Player Prefab**. Unity creates the Lobby player **GameObject** when a client connects (that is, when a player joins the game), and it exists until the client disconnects.

The Network Lobby Player component holds the "ready" state for each player, and handles commands while in the lobby. You can add user **scripts** to the prefab to hold game-specific player data.

The Network Lobby Player component supplies some virtual method callbacks that can be used for custom lobby behaviour. These are:

```
public virtual void OnClientEnterLobby();
public virtual void OnClientExitLobby();
public virtual void OnClientReady(bool readyState);
```

Unity calls the method OnClientEnterLobby on the client when the game enters the lobby. This happens when the lobby Scene starts for the first time, and also when returning to the lobby from the gameplay Scene.

Unity calls the method OnClientExitLobby on the client when the game exits the lobby. This happens when switching to the gameplay Scene.

Unity calls the method OnClientReady on the client when the ready state of that player changes.

Game Player Prefab

A game starts when all players have indicated they are ready. When the game starts, Unity creates a GameObject for each player, based on the the **Game Player Prefab**. Unity destroys these GameObjects at the end of the game, when players when re-enter the lobby. The **Game Player Prefab** handles commands while in the game. This prefab is a standard networked GameObject, and must have a Network Identity component attached.

Minimum Players

On the Network Lobby Manager component, the **Minimum Players** field represents the minimum number of players that need to indicate that they are ready before the game starts. If the number of connected clients reaches the **Minimum Players** value, then waiting for all connected clients to become "Ready" starts the Match.

For example if "Minimum Players" is set to 2:

Start one instance of the game, and begin the game in host mode. Then in your game's lobby interface, press "Start" for your player. You are still in the lobby, because the minimum number of ready players to start a game is 2.
Start two more instances of the game, and begin those games in client mode in those instances. Wait for all connected players (in this example, three) to become ready. Press "Start" in the Lobby **UI** for one player. Now, two players are ready, but still in the lobby. Press "Start" in the Lobby **UI** for the last player. All players move to the main game Scene.

Adding the Lobby to a game

These steps outline the basic process for adding a Network Lobby to a multiplayer game using Unity's built-in **networking** features:

Create a new Scene for the lobby

Add the Scene to the Build Settings (**File** > **Build Settings...** > **Add Open Scenes**), as the first Scene
Create a new GameObject in the new Scene and rename it *LobbyManager*
Add the Network Lobby Manager component to the *LobbyManager* GameObject
Add the **Network Manager** HUD component to the *LobbyManager* GameObject
Open the **Inspector** for the Network Lobby Manager component
In the Network Lobby Manager, set the **Lobby Scene** to the Scene that contains the *LobbyManager* GameObject
In the Network Lobby Manager, set the **Play Scene** to the main gameplay Scene for the game
Create a new GameObject and rename it *LobbyPlayer*
Add the Network Lobby Player component to the *LobbyPlayer*
Create a prefab for the *LobbyPlayer* GameObject, and delete the instance from the Scene
Set the LobbyPlayerPrefab field (in the Network Lobby Manager inspector) to the *LobbyPlayer* prefab
Set the GamePlayerPrefab field (in the Network Lobby Manager inspector) to the prefab for the player in the main game
Save the Scene
Run the game

This version of the Network Lobby Manager is a very simple implementation, and uses a placeholder user interface, much like like the Network Manager HUD. Before you release your game, you should replace this with your own own user interface that matches your game's visual design and feature requirements.

For an example of a better user interface, see the [multiplayer-lobby asset package] ((https://www.assetstore.unity3d.com/en/#!/content/41836) available on the Asset Store.

The NetworkLobbyManager class has many virtual function callbacks for custom lobby behaviour. The most important function is OnLobbyServerSceneLoadedForPlayer, which is called on the server for each player when they transition from the lobby to the main game. This is the ideal place to apply settings from the lobby GameObject to the player GameObject.

```
// for users to apply settings from their lobby player GameObject to their i
public override bool OnLobbyServerSceneLoadedForPlayer(GameObject lobbyPlaye
{
    var cc = lobbyPlayer.GetComponent<ColorControl>();
    var player = gamePlayer.GetComponent<Player>();
    player.myColor = cc.myColor;
    return true;
}
```

Sample Project

There is a sample project on the** Unity **Asset Store** that uses the Network Lobby Manager and provides a GUI for the lobby. You can use this as a starting point for making your own Lobby for your multiplayer game. See Asset Store: Lobby Sample Project.

Using the Transport Layer API

Leave feedback

In addition to the high-level networking API (HLAPI), Unity also provides access to a lower-level **networking** API called the **Transport Layer**. The Transport Layer allows you to build your own **networking** systems with more specific or advanced requirements for your game's **networking**.

The Transport Layer is a thin layer working on top of the operating system's sockets-based networking. It can send and receive messages represented as arrays of bytes, and offers a number of different "quality of service" options to suit different scenarios. It is focused on flexibility and performance, and exposes an API within the NetworkTransport class.

The Transport Layer supports base services for network communication. These base services include:

Establishing connections

Communicating using a variety of "quality of services"

Flow control

Base statistics

Additional services, such as communication via relay server or local discovery

The Transport Layer can use two protocols: UDP for generic communications, and WebSockets for **WebGL**. To use the Transport Layer directly, the typical workflow is as follows:

Initialize the Network Transport Layer

Configure network topology

Create a host

Start communication (handling connections and sending/receiving messages)

Shutdown library after use

See the corresponding sections below to learn about the technical details of each section. Each section provides a code snippet to include in your networking script.

Step 1: Initialize the Network Transport Layer

When initializing the Network Transport Layer, you can choose between the default initialization demonstrated in the code sample below (with no arguments), or you can provide additional parameters which control the overall behaviour of the network layer, such as the maximum packet size and the thread timeout limit.

To initialize the transport layer with default settings, call Init():

```
    // Initializing the Transport Layer with no arguments (default settings)
    NetworkTransport.Init();

To initialize the transport layer with your own configuration just add your configuration a

    // An example of initializing the Transport Layer with custom settings
    GlobalConfig gConfig = new GlobalConfig();
    gConfig.MaxPacketSize = 500;
    NetworkTransport.Init(gConfig);
```

You should only use custom `Init` values if you have an unusual networking environment and are familiar with the specific settings you need. As a rule of thumb, if you are developing a typical multiplayer game to be played across the internet, the default `Init` settings are enough.

Step 2: Configure the network topology

The next step is to configure connections between peers. Network topology defines how many connections allowed and what connection configuration will used. If your game needs to send network messages which vary in importance (eg low importance such as incidental sound effects, vs high importance such as whether a player scored a point), you might want to define several communication channels, each with a different quality of service level specified to suit the specific types of messages that you want to send, and their relative importance within your game.

```
ConnectionConfig config = new ConnectionConfig();
int myReliableChannelId  = config.AddChannel(QosType.Reliable);
int myUnreliableChannelId = config.AddChannel(QosType.Unreliable);
```

The example above defines two communication channels with different quality of service values. QosType.Reliable delivers a message and ensures that the message is delivered, while QosType.Unreliable sends a message faster, but without any checks to ensure it was delivered.

You can also adjust properties on ConnectionConfig to specify configuration settings for each connection. However, when making a connection from one client to another, the settings should be the same for both connected peers, or the connection fails with a CRCMismatch error.

The final step of network configuration is topology definition.

```
HostTopology topology = new HostTopology(config, 10);
```

This example defines host topology as being able to allow up to 10 connections. These connections are the ones you configured in Step 1.

Step 3: Create a host

Now that you have performed the first two preliminary set-up steps, you need to create a host (open socket):

```
int hostId = NetworkTransport.AddHost(topology, 8888);
```

This code example adds a new host on port 8888 and any IP addresses. The host supports up to 10 connections (you configured this in Step 2). These connections are the ones you configured in the Step 1.

Step 4: Start communicating

In order to start communicating, you need to set up a connection to another host. To do this, call `Connect()`. This sets up a connection between you and the remote host. An event is received to indicate whether the connection is successful.

First, connect to the remote host at 192.168.1.42 with port 8888. The assigned `connectionId` is returned.

```
connectionId = NetworkTransport.Connect(hostId, "192.168.1.42", 8888, 0, out error);
```

When the connection is done, a ConnectEvent is received. Now you can start sending data.

```
NetworkTransport.Send(hostId, connectionId, myReliableChannelId, buffer, bufferLength,  out
```

When you are done with a connection, call `Disconnect()` to disconnect the host.

```
NetworkTransport.Disconnect(hostId, connectionId, out error);
```

To check if your function calls were successful, you can cast the `out error` to a NetworkError. NetworkEror.Ok indicates that no errors have been encountered.

To check host status, you can use two functions:

For polling events off the internal event queue you can call either

NetworkTransport.Receive(out recHostId, out connectionId, out channelId, recBuffer, bufferSize, out dataSize, out error);

Or NetworkTransport.ReceiveFromHost(recHostId, out connectionId, out channelId, recBuffer, bufferSize, out dataSize, out error);

Both of these functions return events from the queue; the first function will return events from any host, and the recHostId variable will be assigned with the host id that the message comes from, whereas the second function returns events only from the host specified by the recHostId that you provide.

One way to poll data from `Receive` is to call it in your `Update` function;

```
void Update()
{
```

```
    int recHostId;
    int connectionId;
    int channelId;
    byte[] recBuffer = new byte[1024];
    int bufferSize = 1024;
    int dataSize;
    byte error;
    NetworkEventType recData = NetworkTransport.Receive(out recHostId, out connectionId, ou
    switch (recData)
    {
        case NetworkEventType.Nothing:                    break;
        case NetworkEventType.ConnectEvent:               break;
        case NetworkEventType.DataEvent:                  break;
        case NetworkEventType.DisconnectEvent:           break;

  case NetworkEventType.BroadcastEvent:

       break;
    }
}
```

There are 5 type of events that you can receive.

NetworkEventType.Nothing: The event queue has nothing to report.

NetworkEventType.ConnectEvent : You have received a connect event. This can be either a successful connect request, or a connection response.

```
case NetworkEventType.ConnectEvent:
    if(myConnectionId == connectionId)
        //my connect request was approved
    else
        //somebody else sent a connect request to me
    break;
```

NetworkEventType.DataEvent: You have received a data event. You receive a data event when there is some data ready to be recieved. If the `recBuffer` is big enough to contain data, data is copied into the buffer. If not, the event contains a MessageToLong network error. If this happens, you need to reallocate the buffer to a larger size and call the `DataEvent` function again.

NetworkEventType.DisconnectEvent: Your established connection has disconnected, or your connect request has failed. Check the error code to find out why this has happened.

```
case NetworkEventType. DisconnectEvent:
    if(myConnectionId == connectionId)
        //cannot connect for some reason, see error
    else
        //one of the established connections has disconnected
    break;
```

NetworkEventType.BroadcastEvent: Indicates that you have received a broadcast event, and you can now call GetBroadcastConnectionInfo and GetBroadcastConnectionMessage to retrieve more information.

WebGL support

You can use WebSockets on WebGL, however web clients can only connect to hosts, they cannot be a host themselves. This means the host must be a standalone player (Win, Mac or Linux only). For client-side configuration, all steps described above (including topology and configuration) are the same. On the server, call the following:

```
NetworkTransport.AddWebsocketHost(topology, port, ip);
```

The IP address above should be the specific address you want to listen on, or you can pass null as the IP address if you want the host to listen all network interfaces.

A server can only support only one WebSocket host at a time, but it can handle other generic hosts at the same time:

```
NetworkTransport.AddWebsocketHost(topology, 8887, null);
NetworkTransport.AddHost(topology, 8888);
```

NetworkReader and NetworkWriter serializers

Leave feedback

Use the NetworkReader and NetworkWriter classes to write data to byte streams.

The Multiplayer High Level API is built using these classes, and uses them extensively. However, you can use them directly if you want to implement your own custom transport functionality. They have specific serialization functions for many Unity types (See NetworkWriter.Write for the full list of types).

To use the classes, create a writer instance, and write individual variables into it. These are serialized internally into a byte array, and this can be sent over the network. On the receiving side it's important that the reader instance for the byte array reads back the variables in exactly the same order they were written in.

This can be used with the MessageBase class to make byte arrays that contain serialized network messages.

```
void SendMessage(short msgType, MessageBase msg, int channelId)
{
    // write the message to a local buffer
    NetworkWriter writer = new NetworkWriter();
    writer.StartMessage(msgType);
    msg.Serialize(writer);
    writer.FinishMessage();

    myClient.SendWriter(writer, channelId);
}
```

This message is correctly formatted so that a message handler function can be invoked for it.

Using the NetworkReader and NetworkWriter classes with the NetworkServerSimple and NetworkClient classes

The following code sample is a rather low level demonstration, using the lowest level classes from the high-level API for setting up connectivity.

This is the code for connecting the client and server together:

```
using UnityEngine;
using UnityEngine.Networking;
public class Serializer : MonoBehaviour {
    NetworkServerSimple m_Server;
    NetworkClient m_Client;
    const short k_MyMessage = 100;
```

```
    // When using a server instance like this it must be pumped manually
    void Update() {
        if (m_Server != null)
            m_Server.Update();
    }

    void StartServer() {
        m_Server = new NetworkServerSimple();
        m_Server.RegisterHandler(k_MyMessage, OnMyMessage);
        if (m_Server.Listen(5555))
            Debug.Log("Started listening on 5555");
    }

    void StartClient() {
        m_Client = new NetworkClient();
        m_Client.RegisterHandler(MsgType.Connect, OnClientConnected);
        m_Client.Connect("127.0.0.1", 5555);
    }

    void OnClientConnected(NetworkMessage netmsg) {
        Debug.Log("Client connected to server");
        SendMessage();
    }
}
```

The next piece of code sends a message using the network reader and network writer, but uses the message handlers built into these classes:

void SendMessage() { NetworkWriter writer = new NetworkWriter(); writer.StartMessage(k_MyMessage); writer.Write(42); writer.Write("What is the answer"); writer.FinishMessage(); m_Client.SendWriter(writer, 0); }

void OnMyMessage(NetworkMessage netmsg) { Debug.Log("Got message, size=" + netmsg.reader.Length); var someValue = netmsg.reader.ReadInt32(); var someString = netmsg.reader.ReadString(); Debug.Log("Message value=" + someValue + " Message string="' + someString + '"'); }

When setting up messages for the message handlers, you should always use `NetworkWriter.StartMessage()` (with the message type ID) and NetworkWriter.FinishMessage() calls. When not using the byte arrays, you can skip that step.

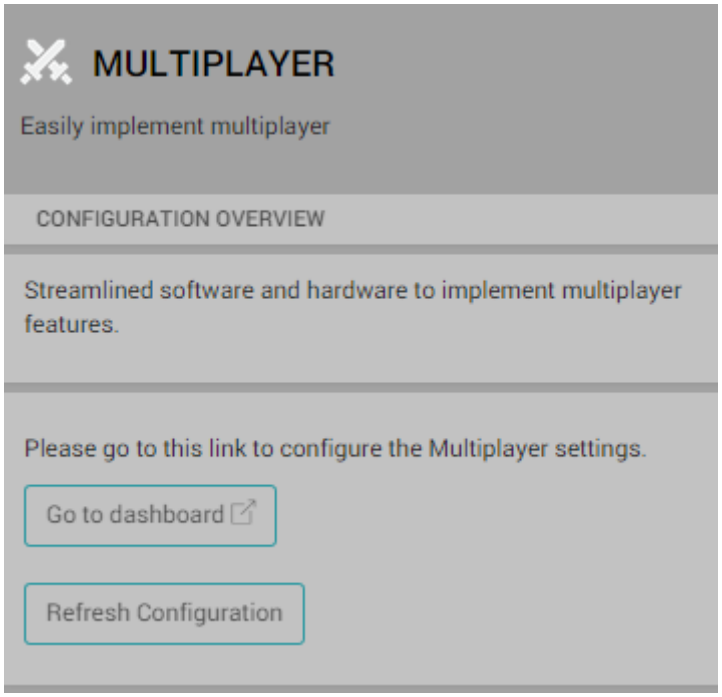
Setting up Unity Multiplayer

Leave feedback

To start using Unity Multiplayer, your project must be set up to use Unity Services. Once you have done this, you can enable the Multiplayer Service.

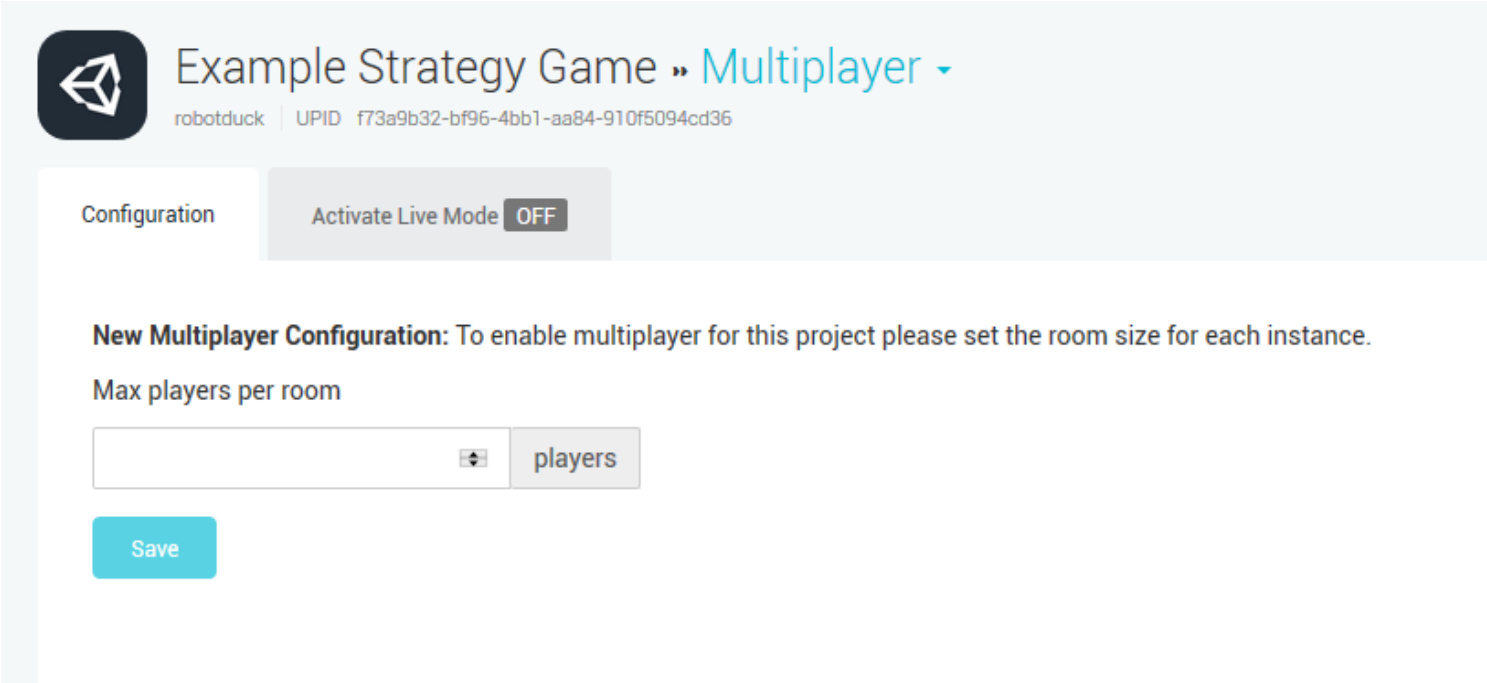
To do this, open the **Services** window by selecting **Window** > **General** > **Services** in the menu bar. In the Services window, select **Multiplayer**.

This opens the Multiplayer Services window.

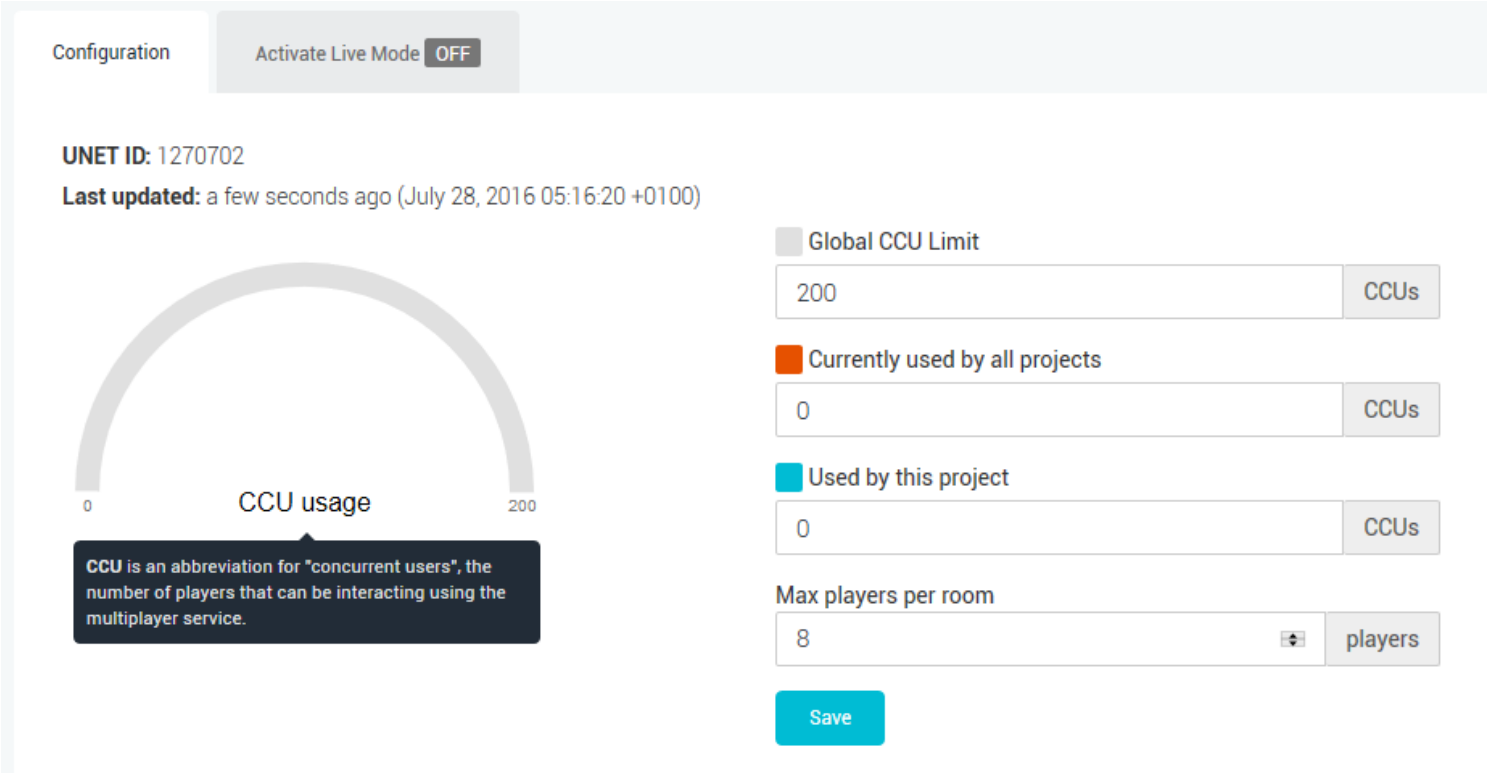


The **Go To Dashboard** button takes you to the web-based Services Dashboard, where you can set up the Multiplayer configuration for your project.

If you haven't set up your project with the Multiplayer service yet, you are prompted to set up a **New Multiplayer Configuration**. To do this, enter the number of players you want per room and click **Save**.



Once you have clicked Save, the Multiplayer Services Dashboard reflects your current project.



You are now ready to integrate your project with Unity Multiplayer!

Integrating the Multiplayer Service

Leave feedback

There are three different methods you can use to start working with the Multiplayer Service in your project. These three methods give you a different level of control depending on your needs.

using NetworkManagerHUD. (simplest, requires no scripting)
using NetworkServer and NetworkClient. (high-level, simpler scripting)
using NetworkTransport directly. (low-Level, more complex scripting)

The first method, using **NetworkManagerHUD** offers the highest level of abstraction, meaning that the Service does most of the work for you. This is therefore the simplest method to use, and most suitable for those new to creating multiplayer games. It provides a simple graphical interface which you can use to perform the basic multiplayer tasks of creating, listing, joining and starting games (referred to as 'matches').

The second method, using **NetworkServer** and **NetworkClient**, uses our Networking High-Level API to do these same tasks. This method is more flexible; you can use the examples provided to integrate the basic multiplayer tasks into your games own **UI**.

The third method, using *NetworkTransport* directly, gives you maximum control, but is only usually necessary if you have unusual requirements which are not met by using our Networking High-Level API.

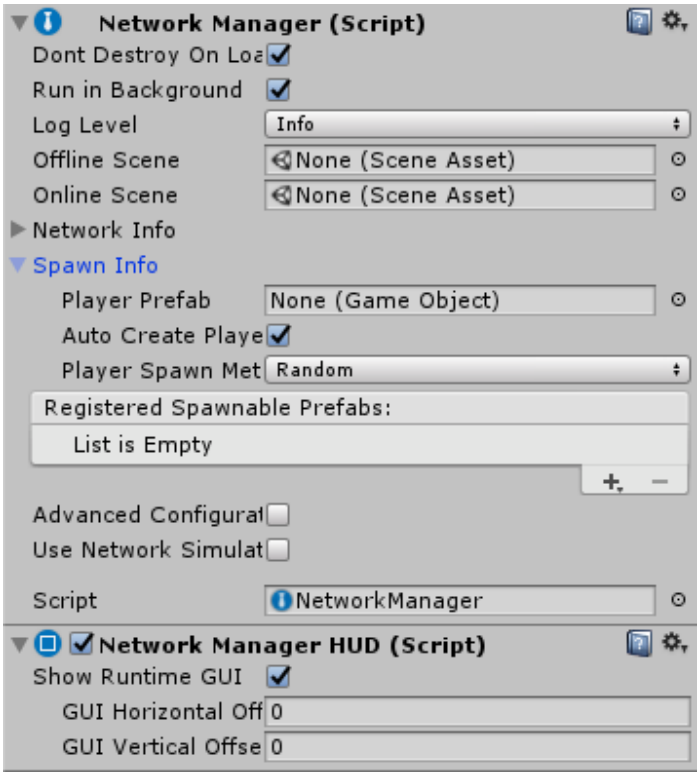
Integration using the HUD

Leave feedback

To integrate Unity Multiplayer Services using the **NetworkManagerHUD**, follow these steps:

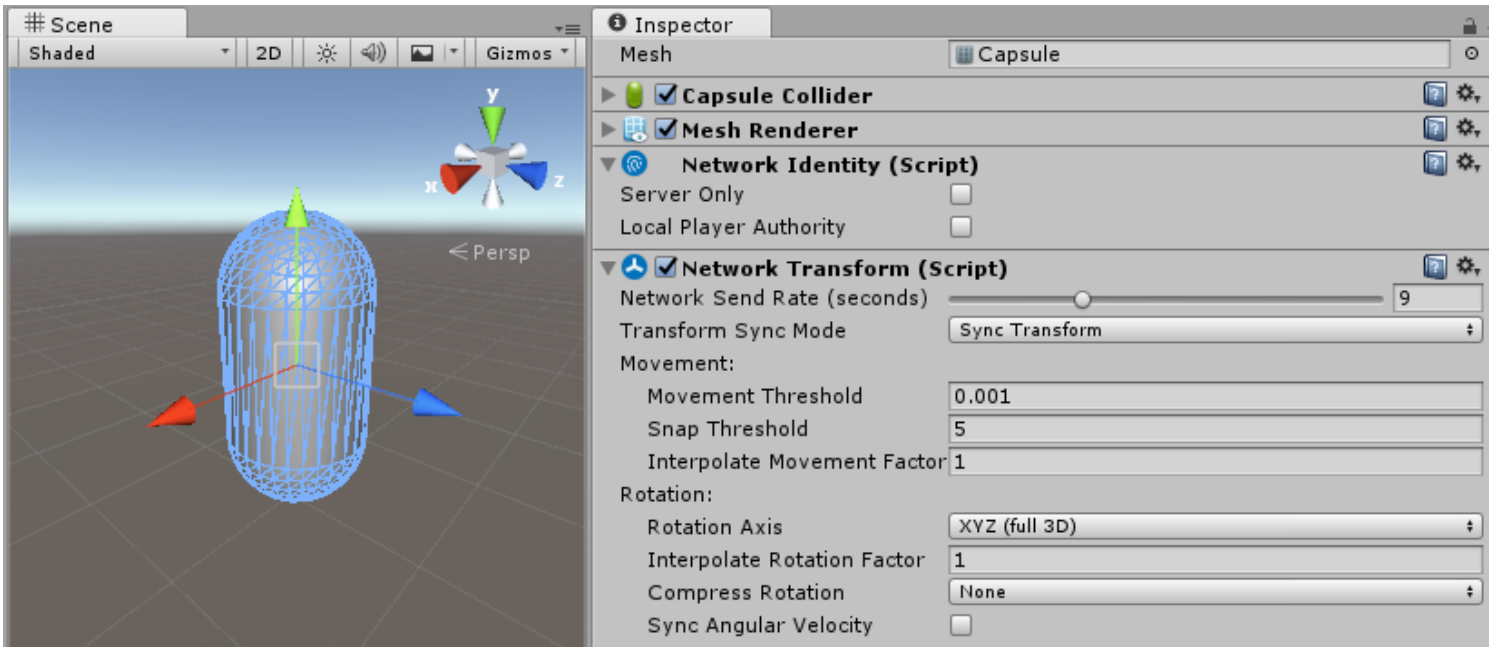
Create an empty **GameObject** in your **Scene**.

Add the **components** **NetworkManager** and **NetworkManagerHUD** to the empty GameObject. Rename this object to “Network Manager” so that you know what it is.

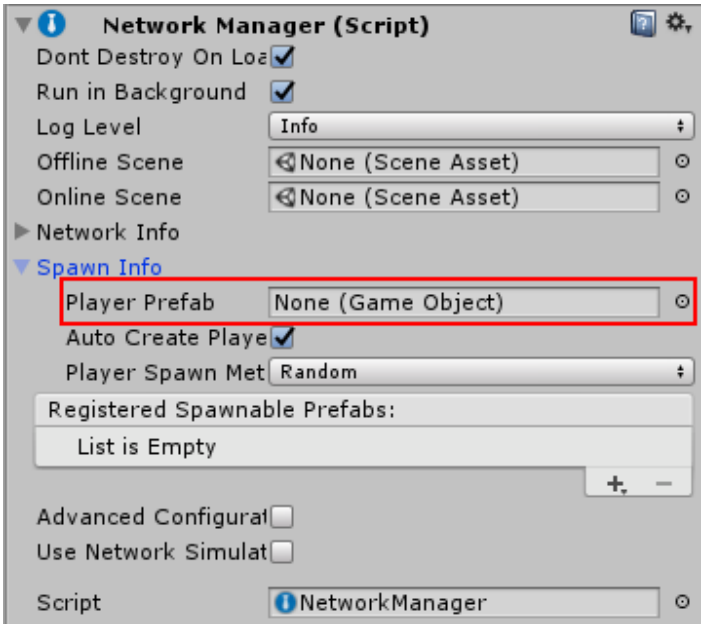


Create a **prefab** to represent your player. Players connected to your game will each control an instance of this **prefab**.

Add the **NetworkIdentity** and **NetworkTransform** component to your player prefab. The **NetworkTransform** component synchronizes the player GameObject’s movement. If you’re making a game where players don’t move, you don’t need this.

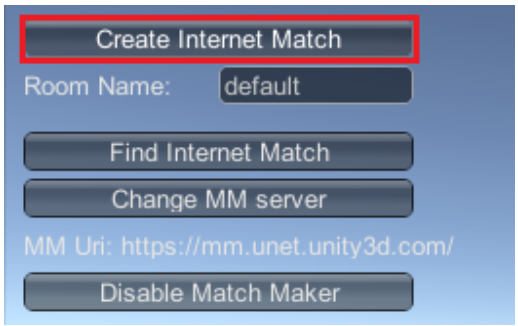


Add your player prefab to the the **Network Manager**'s **Player Prefab** property in the **inspector**.

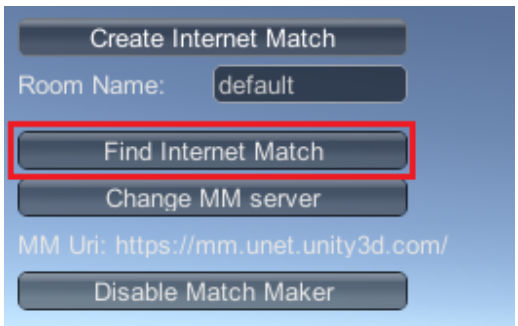


Build and run your project. The Network Manager HUD shows an in-game menu. Click **Enable Match Maker**.

Choose a room name and click **Create Internet Match** on the hosting application.



Run more instances of your project, and click **Find Internet Match** on these clients. Your room name should now appear.



Click **Join Match**. Your players should now be connected to the same match.

Integration using Unity's High-Level API

Leave feedback

To integrate Unity Multiplayer Services using the Networking High-Level API, you must use the NetworkMatch class directly in your **scripts**. To use it, you have to call the functions in `NetworkMatch` manually and handle the callbacks yourself.

Below is an example of how you can create a match, list matches, and join a match using only the NetworkMatch, NetworkServer and NetworkClient classes.

This script sets up the matchmaker to point to the public Unity matchmaker server. It calls the `NetworkMatch` functions for creating, listing, and joining matches:

CreateMatch to create a match
JoinMatch to join a match
ListMatches for listing matches registered on the matchmaker server

Internally, `NetworkMatch` uses web services to establish a match, and the given callback function is invoked when the process is complete, such as `OnMatchCreate` for match creation.

```
using UnityEngine;
using UnityEngine.Networking;
using UnityEngine.Networking.Match;
using System.Collections.Generic;

public class HostGame : MonoBehaviour
{
    List<MatchInfoSnapshot> matchList = new List<MatchInfoSnapshot>();
    bool matchCreated;
    NetworkMatch networkMatch;

    void Awake()
    {
        networkMatch = gameObject.AddComponent<NetworkMatch>();
    }

    void OnGUI()
    {
        // You would normally not join a match you created yourself but this is
        if (GUILayout.Button("Create Room"))
        {
            string matchName = "room";
            uint matchSize = 4;
            bool matchAdvertise = true;
            string matchPassword = "";
```

```
            networkMatch.CreateMatch(matchName, matchSize, matchAdvertise, match
        }

        if (GUILayout.Button("List rooms"))
        {
            networkMatch.ListMatches(0, 20, "", true, 0, 0, OnMatchList);
        }

        if (matchList.Count > 0)
        {
            GUILayout.Label("Current rooms");
        }
        foreach (var match in matchList)
        {
            if (GUILayout.Button(match.name))
            {
                networkMatch.JoinMatch(match.networkId, "", "", "", 0, 0, OnMatc
            }
        }
    }

    public void OnMatchCreate(bool success, string extendedInfo, MatchInfo match
    {
        if (success)
        {
            Debug.Log("Create match succeeded");
            matchCreated = true;
            NetworkServer.Listen(matchInfo, 9000);
            Utility.SetAccessTokenForNetwork(matchInfo.networkId, matchInfo.acce
        }
        else
        {
            Debug.LogError("Create match failed: " + extendedInfo);
        }
    }

    public void OnMatchList(bool success, string extendedInfo, List<MatchInfoSna
    {
        if (success && matches != null && matches.Count > 0)
        {
            networkMatch.JoinMatch(matches[0].networkId, "", "", "", 0, 0, OnMat
        }
        else if (!success)
        {
            Debug.LogError("List match failed: " + extendedInfo);
        }
```

```
    }

    public void OnMatchJoined(bool success, string extendedInfo, MatchInfo match
    {
        if (success)
        {
            Debug.Log("Join match succeeded");
            if (matchCreated)
            {
                Debug.LogWarning("Match already set up, aborting...");
                return;
            }
            Utility.SetAccessTokenForNetwork(matchInfo.networkId, matchInfo.acce
            NetworkClient myClient = new NetworkClient();
            myClient.RegisterHandler(MsgType.Connect, OnConnected);
            myClient.Connect(matchInfo);
        }
        else
        {
            Debug.LogError("Join match failed " + extendedInfo);
        }
    }

    public void OnConnected(NetworkMessage msg)
    {
        Debug.Log("Connected!");
    }
}
```

Integration using NetworkTransport

Leave feedback

If you need the maximum amount of flexibilty when integrating Unity Multiplayer Services, you can use the `NetworkTransport` class directly. This method requires more code, but allows you to control every detail of how your game integrates with the Multiplayer Services.

This is an example of how you can connect directly using the `NetworkTransport` class:

```
using UnityEngine;
using UnityEngine.Networking;
using UnityEngine.Networking.Types;
using UnityEngine.Networking.Match;
using System.Collections.Generic;

public class DirectSetup : MonoBehaviour
{
    // Matchmaker related
    List<MatchInfoSnapshot> m_MatchList = new List<MatchInfoSnapshot>();
    bool m_MatchCreated;
    bool m_MatchJoined;
    MatchInfo m_MatchInfo;
    string m_MatchName = "NewRoom";
    NetworkMatch m_NetworkMatch;

    // Connection/communication related
    int m_HostId = -1;
    // On the server there will be multiple connections, on the client this will
    List<int> m_ConnectionIds = new List<int>();
    byte[] m_ReceiveBuffer;
    string m_NetworkMessage = "Hello world";
    string m_LastReceivedMessage = "";
    NetworkWriter m_Writer;
    NetworkReader m_Reader;
    bool m_ConnectionEstablished;

    const int k_ServerPort = 25000;
    const int k_MaxMessageSize = 65535;

    void Awake()
    {
        m_NetworkMatch = gameObject.AddComponent<NetworkMatch>();
    }

    void Start()
    {
        m_ReceiveBuffer = new byte[k_MaxMessageSize];
```

```
    m_Writer = new NetworkWriter();
    // While testing with multiple standalone players on one machine this wi
    Application.runInBackground = true;
}

void OnApplicationQuit()
{
    NetworkTransport.Shutdown();
}

void OnGUI()
{
    if (string.IsNullOrEmpty(Application.cloudProjectId))
        GUILayout.Label("You must set up the project first. See the Multipla
    else
        GUILayout.Label("Cloud Project ID: " + Application.cloudProjectId);

    if (m_MatchJoined)
        GUILayout.Label("Match joined '" + m_MatchName + "' on Matchmaker se
    else if (m_MatchCreated)
        GUILayout.Label("Match '" + m_MatchName + "' created on Matchmaker s

    GUILayout.Label("Connection Established: " + m_ConnectionEstablished);

    if (m_MatchCreated || m_MatchJoined)
    {
        GUILayout.Label("Relay Server: " + m_MatchInfo.address + ":" + m_Mat
        GUILayout.Label("NetworkID: " + m_MatchInfo.networkId + " NodeID: "
        GUILayout.BeginHorizontal();
        GUILayout.Label("Outgoing message:");
        m_NetworkMessage = GUILayout.TextField(m_NetworkMessage);
        GUILayout.EndHorizontal();
        GUILayout.Label("Last incoming message: " + m_LastReceivedMessage);

        if (m_ConnectionEstablished && GUILayout.Button("Send message"))
        {
            m_Writer.SeekZero();
            m_Writer.Write(m_NetworkMessage);
            byte error;
            for (int i = 0; i < m_ConnectionIds.Count; ++i)
            {
                NetworkTransport.Send(m_HostId,
                    m_ConnectionIds[i], 0, m_Writer.AsArray(), m_Writer.Posi
                if ((NetworkError)error != NetworkError.Ok)
                    Debug.LogError("Failed to send message: " + (NetworkErro
            }
```

```
        }

        if (GUILayout.Button("Shutdown"))
        {
            m_NetworkMatch.DropConnection(m_MatchInfo.networkId,
                m_MatchInfo.nodeId, 0, OnConnectionDropped);
        }
    }
    else
    {
        if (GUILayout.Button("Create Room"))
        {
            m_NetworkMatch.CreateMatch(m_MatchName, 4, true, "", "", "", 0,
        }

        if (GUILayout.Button("Join first found match"))
        {
            m_NetworkMatch.ListMatches(0, 1, "", true, 0, 0, (success, info,
            {
                if (success && matches.Count > 0)
                    m_NetworkMatch.JoinMatch(matches[0].networkId, "", "", "
            });
        }

        if (GUILayout.Button("List rooms"))
        {
            m_NetworkMatch.ListMatches(0, 20, "", true, 0, 0, OnMatchList);
        }

        if (m_MatchList.Count > 0)
        {
            GUILayout.Label("Current rooms:");
        }
        foreach (var match in m_MatchList)
        {
            if (GUILayout.Button(match.name))
            {
                m_NetworkMatch.JoinMatch(match.networkId, "", "", "", 0, 0,
            }
        }
    }
}

public void OnConnectionDropped(bool success, string extendedInfo)
{
    Debug.Log("Connection has been dropped on matchmaker server");
```

```
        NetworkTransport.Shutdown();
        m_HostId = -1;
        m_ConnectionIds.Clear();
        m_MatchInfo = null;
        m_MatchCreated = false;
        m_MatchJoined = false;
        m_ConnectionEstablished = false;
    }

    public virtual void OnMatchCreate(bool success, string extendedInfo, MatchIn
    {
        if (success)
        {
            Debug.Log("Create match succeeded");
            Utility.SetAccessTokenForNetwork(matchInfo.networkId, matchInfo.acce

            m_MatchCreated = true;
            m_MatchInfo = matchInfo;

            StartServer(matchInfo.address, matchInfo.port, matchInfo.networkId,
                matchInfo.nodeId);
        }
        else
        {
            Debug.LogError("Create match failed: " + extendedInfo);
        }
    }

    public void OnMatchList(bool success, string extendedInfo, List<MatchInfoSna
    {
        if (success && matches != null)
        {
            m_MatchList = matches;
        }
    else if(!success)
    {
        Debug.LogError("List match failed: " + extendedInfo);
    }
    }

    // When we've joined a match we connect to the server/host
    public virtual void OnMatchJoined(bool success, string extendedInfo, MatchIn
    {
        if (success)
        {
            Debug.Log("Join match succeeded");
```

```
        Utility.SetAccessTokenForNetwork(matchInfo.networkId, matchInfo.acce

        m_MatchJoined = true;
        m_MatchInfo = matchInfo;

        Debug.Log("Connecting to Address:" + matchInfo.address +
            " Port:" + matchInfo.port +
            " NetworKID: " + matchInfo.networkId +
            " NodeID: " + matchInfo.nodeId);
        ConnectThroughRelay(matchInfo.address, matchInfo.port, matchInfo.net
            matchInfo.nodeId);
    }
    else
    {
        Debug.LogError("Join match failed: " + extendedInfo);
    }
}

void SetupHost(bool isServer)
{
    Debug.Log("Initializing network transport");
    NetworkTransport.Init();
    var config = new ConnectionConfig();
    config.AddChannel(QosType.Reliable);
    config.AddChannel(QosType.Unreliable);
    var topology = new HostTopology(config, 4);
    if (isServer)
        m_HostId = NetworkTransport.AddHost(topology, k_ServerPort);
    else
        m_HostId = NetworkTransport.AddHost(topology);
}

void StartServer(string relayIp, int relayPort, NetworkID networkId, NodeID
{
    SetupHost(true);

    byte error;
    NetworkTransport.ConnectAsNetworkHost(
        m_HostId, relayIp, relayPort, networkId, Utility.GetSourceID(), node
}

void ConnectThroughRelay(string relayIp, int relayPort, NetworkID networkId,
{
    SetupHost(false);

    byte error;
```

```
    NetworkTransport.ConnectToNetworkPeer(
        m_HostId, relayIp, relayPort, 0, 0, networkId, Utility.GetSourceID()
}

void Update()
{
    if (m_HostId == -1)
        return;

    var networkEvent = NetworkEventType.Nothing;
    int connectionId;
    int channelId;
    int receivedSize;
    byte error;

    // Get events from the relay connection
    networkEvent = NetworkTransport.ReceiveRelayEventFromHost(m_HostId, out
    if (networkEvent == NetworkEventType.ConnectEvent)
        Debug.Log("Relay server connected");
    if (networkEvent == NetworkEventType.DisconnectEvent)
        Debug.Log("Relay server disconnected");

    do
    {
        // Get events from the server/client game connection
        networkEvent = NetworkTransport.ReceiveFromHost(m_HostId, out connec
            m_ReceiveBuffer, (int)m_ReceiveBuffer.Length, out receivedSize,
        if ((NetworkError)error != NetworkError.Ok)
        {
            Debug.LogError("Error while receiveing network message: " + (Net
        }

        switch (networkEvent)
        {
            case NetworkEventType.ConnectEvent:
                {
                    Debug.Log("Connected through relay, ConnectionID:" + con
                        " ChannelID:" + channelId);
                    m_ConnectionEstablished = true;
                    m_ConnectionIds.Add(connectionId);
                    break;
                }
            case NetworkEventType.DataEvent:
                {
                    Debug.Log("Data event, ConnectionID:" + connectionId +
                        " ChannelID: " + channelId +
```

```
                        " Received Size: " + receivedSize);
                    m_Reader = new NetworkReader(m_ReceiveBuffer);
                    m_LastReceivedMessage = m_Reader.ReadString();
                    break;
                }
            case NetworkEventType.DisconnectEvent:
                {
                    Debug.Log("Connection disconnected, ConnectionID:" + con
                    break;
                }
            case NetworkEventType.Nothing:
                break;
        }
    } while (networkEvent != NetworkEventType.Nothing);
  }
}
```

Common Errors

Leave feedback

Two common errors you might encounter are:

```
Create match failed: Failed to access NetworkDB appId=0
```

This occurs if you have not linked the project with Unity Services. To link your project with Unity Services, see Setting up your project for Unity Services.

```
Create match failed: Failed because of invalid projectid setup. Please verify project setup at https://multiplayer.unity3d.com
```

This occurs if you have not enabled the Multiplayer Service. To fix this, make sure you have set up your project for Multiplayer Services correctly.

Customized Matchmaking callbacks

Leave feedback

When using the Network Manager with the Network Manager HUD, the NetworkManager.StartMatchmaker method enables matchmaking, and populates the NetworkManager.matchMaker property with a NetworkMatch object. Once this is active, the Network Manager HUD uses it and calls methods on NetworkManager to let you perform simple matchmaking.

There are virtual functions on `NetworkManager` that you can customize by deriving your own class from `NetworkManager`. You can then customize the way your new `NetworkManager` class responds to Matchmaker callbacks.

Here are the callbacks and their default implementations. If you override them, there are some methods which require that you call the base implementation, otherwise the functionality with the Network Manager HUD breaks. For example, the default implementation of `OnMatchCreate`, starts the host.

```
using UnityEngine;
using UnityEngine.Networking;
using UnityEngine.Networking.Match;

public class CustomManager : NetworkManager {

    public override void OnMatchCreate(bool success, string extendedInfo, MatchI
        if (LogFilter.logDebug) { Debug.LogFormat("NetworkManager OnMatchCreate
        if(success)
            StartHost(matchInfo);
    }

    public override void OnMatchJoined(bool success, string extendedInfo, MatchI
        if (LogFilter.logDebug) { Debug.LogFormat("NetworkManager OnMatchJoined
        if(success)
            StartClient(matchInfo);
    }

    public override void OnMatchList(bool success, string extendedInfo, List<Mat
        if (LogFilter.logDebug) { Debug.LogFormat("NetworkManager OnMatchList Su
        matches = matchList;
    }

    public override void OnDestroyMatch(bool success, string extendedInfo) {
        if (LogFilter.logDebug) { Debug.LogFormat("NetworkManager OnDestroyMatch
    }

    public override void OnDropConnection(bool success, string extendedInfo) {
        if (LogFilter.logDebug) { Debug.LogFormat("NetworkManager OnDestroyMatch
    }
```

```
    public override void OnSetMatchAttributes(bool success, string extendedInfo)
        if (LogFilter.logDebug) { Debug.LogFormat("NetworkManager OnDestroyMatch
    }
}
```

Networking Tips for Mobile devices.

Leave feedback

Unity's Multiplayer system is fully compatible across Desktop, **iOS** and Android devices, players on all these platforms can play your multiplayer game together.

You might want to implement special measures if your game is mainly to be used with Wi-Fi or cellular networks. On some mobile devices, the **networking** chip might be a performance bottleneck, because pings between mobile devices (or between mobile devices and desktops) can be about 40–60ms, even on high-performance Wi-Fi network, and you may observe some delays of over 200ms, despite a low average ping on high performance wifi.

For players to play your game simultaneously from both desktop and mobile platforms (over Wi-Fi or cellular networks), your game server should have a public IP address accessible through the internet, or use the Matchmaker and Relay services.

Note: EDGE / 3G data connections go to sleep very quickly when no data is sent. Therefore, you might sometimes need to "wake" the connection. If you are using the WWW class make sure you connect to your site (and yield until it finishes) before making the Unity networking connection.

Multiplayer Component Reference

Leave feedback

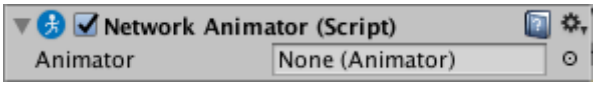
This section gives details of the **components** used with the Unity multiplayer system.

Network Animator

Leave feedback

The Network Animator **component** allows you to synchronize animation states for networked objects. It synchronizes state and parameters from an AnimatorController.

Note that if you create a Network **Animator component** on an empty **GameObject**, Unity also creates a Network Identity component and an Animator component on that GameObject.



The Network Animator component in the Inspector window

Property	**Function
Animator	Use this field to define the Animator component you want the Network Animator to synchronize with.

Details

The Network Animator ensures the synchronisation of GameObject animation across the network - meaning that all players see the animation happen at the same. There are two kinds of authority for networked animation (see documentation on Network system concepts for more information about authority)):

If the GameObject has authority on the client, you should animate it locally on the client that owns the GameObject. That client sends the animation state information to the server, which broadcasts it to all the other clients. For example, this would be suitable for player characters.

If the GameObject has authority on the server, then you should animate it on the server. The server then sends state information to all clients. This is common for animated GameObjects that are not related to a specific client, such as non-player characters.

The Network Animator synchronizes the **animation parameters** checked in the **Inspector** window. It does not automatically synchronize animation triggers. A GameObject with authority can use the function SetTrigger to fire an animation trigger on other clients.

The GetParameterAutoSend and SetParameterAutoSend functions can be used to control which individual animator parameters should be automatically synchronized.

Network Discovery

Leave feedback

Network Discovery is a **component** that allows Unity applications using the **networking** system to find each other on a local network (LAN). This does not allow discovery for internet play. Use the Multiplayer Service (along with MatchMaker and Relay service) to connect to .

The Network Discovery component does not require any integration with Unity services, and is intended as a fully stand-alone solution to finding other games on your local network to connect with.

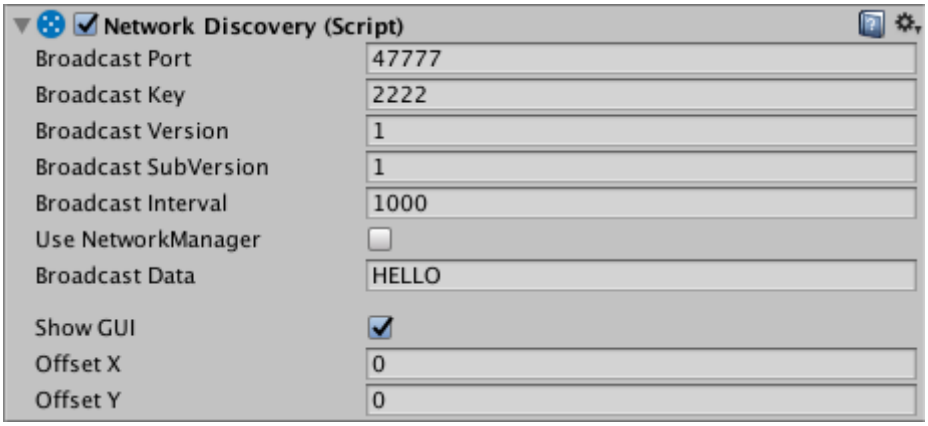


image alt text

The Network Discovery component in the **Inspector** window

Property	Function
Broadcast Port	The network port to broadcast on and listen to.
Broadcast Key	The key to broadcast. This should be a unique value to indicate your discovery's compatibility to other Network Discovery instances. Unique broadcast keys avoid games of different types from trying to connect to each other if they run on the same local network.
Broadcast Version	The major version to include in the broadcast. Use this along with Broadcast SubVersion to indicate version compatibility.
Broadcast SubVersion	The minor version to include in the broadcast. Use this alongside Broadcast Version to indicate version compatibility.
Broadcast Interval	Specify how often Unity should broadcast discovery information, in seconds.
Use NetworkManager	Enable this to use the **Network Manager** settings for broadcasting, and to then auto-join found games.
Broadcast Data	Enter custom data to include in the broadcast. The Network Manager overrides this if you have enabled Use NetworkManager.
Show GUI	Enable this to show the default broadcast GUI in Play mode. This GUI is only intended for developer testing.
Offset X	The x-axis offset of the broadcast GUI. This setting is only visible if Show GUI is enabled.
Offset Y	The y-axis offset of the broadcast GUI. This setting is only visible if Show GUI is enabled.

When running in play mode

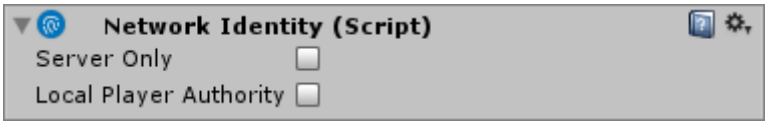
When running in play mode, the following information is also visible in the inspector:

Property	Function
hostId	The host Id being used to broadcast.
running	True if currently broadcasting.
isServer	True if broadcasting as a server.
isClient	True if listening for broadcasts as a client.
broadcastsReceived	A list of broadcast messages received.

Network Identity

Leave feedback

The Network Identity **component** is at the heart of the Unity **networking** high-level API. It controls a **GameObject**'s unique identity on the network, and it uses that identity to make the **networking** system aware of the **GameObject**.



The Network Identity component in the Inspector window

Property	Function
Server Only	Tick this checkbox to ensure that Unity only spawns the GameObject on the server, and not on clients.
Local Player Authority	Tick this checkbox to give authoritative network control of this GameObject to the client that owns it. The player GameObject on that client has authority over it. Other components such as Network Transform use this to determine which client to treat as the source of authority.

Instantiated network GameObjects

With the Unity's server-authoritative networking system, the server must spawn networked GameObjects with network identities, using NetworkServer.Spawn. This automatically creates them on clients that are connected to the server, and assigns them a NetworkInstanceId.

You must put a Network Identity component on any **Prefabs** that spawn at runtime for the network system to use them. See Object Spawning for more information.

Scene-based network GameObjects

You can also network GameObjects that are saved as part of your **Scene** (for example, environmental props). Networking GameObjects makes them behave slightly differently, because you need to have them spawn across the network.

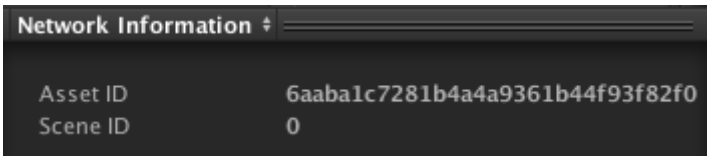
When building your game, Unity disables all Scene-based GameObjects with Network Identity components. When a client connects to the server, the server sends spawn messages to tell the client which Scene GameObjects to enable and what their most up-to-date state information is. This ensures the client's game does not contain GameObjects at incorrect locations when they start playing, or that Unity does not spawn and immediately destroy GameObjects on connection (for example, if an event removed the GameObject before that client connected). See Networked Scene GameObjects for more information.

Network Information in the preview pane

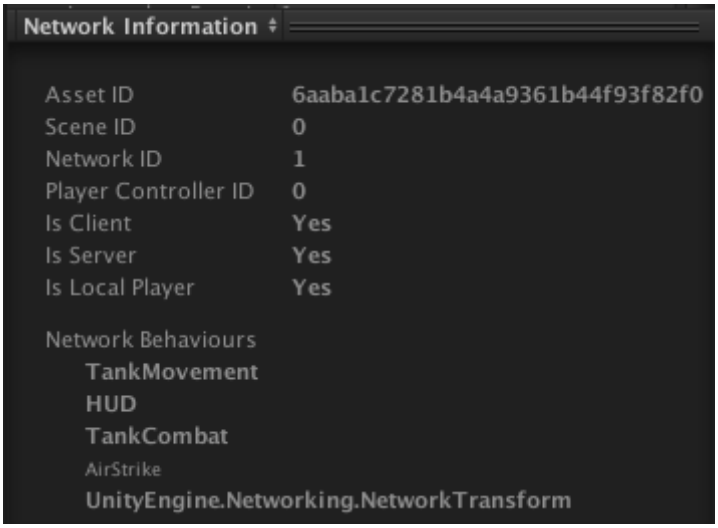
This component contains network tracking information, and displays that information in the preview pane. For example, the scene ID, network ID and **asset** ID the object has been assigned. This allows you to inspect the information which can be useful for investigation and debugging.

The scene ID is valid in all scene objects with a **NetworkIdentity** component. The network ID is the ID of this particular object instance. There might be multiple objects instantiated from a particular prefab, and the network ID is used to identity which object a network update should be applied to. The asset ID refers to which source

asset the object was instantiated from. This is used internally when a particular GameObject prefab is spawned over the network.



At runtime there is more information to display here (a disabled NetworkBehaviour is displayed non-bold):



Preview Pane Information

Property	Description
assetId	This identifies the prefab associated with this object (for spawning).
clientAuthorityOwner	The client that has authority for this object. This will be null if no client has authority.
connectionToClient	The NetworkConnection associated with this NetworkIdentity. This is only valid for **player objects** on the server.
connectionToServer	The NetworkConnection associated with this NetworkIdentity. This is only valid for player objects on a local client.
hasAuthority	True if this object is the authoritative version of the object. This would mean either on a the server for normal objects, or on the client with localPlayerAuthority.
isClient	True if this object is running on a client.
isLocalPlayer	This returns true if this object is the one that represents the player on the local machine.
isServer	True if this object is running on the server, and has been spawned.
localPlayerAuthority	True if this object is controlled by the client that owns it - the local player object on that client has authority over it. This is used by other components such as **NetworkTransform**.
netId	A unique identifier for this network session, assigned when spawned.
observers	The list of client NetworkConnections that are able to see this object. This is read-only.

Property	Description
playerControllerId	The identifier of the controller associated with this object. Only valid for player objects.
SceneId	A unique identifier for networked objects in a Scene. This is only populated in play-mode.
serverOnly	A flag to make this object not be spawned on clients.

Network Lobby Manager

Leave feedback

The NetworkLobbyManager is a specialized type of NetworkManager that provides a multiplayer lobby before entering the main play **scene** of the game. It allows you to set up a network with:

A maximum player limit
Automatic start when all players are ready
Option to prevent players from joining a game in progress
Support for "Couch Multiplayer" (i.e. multiple players per client)
Customizable ways for players to choose options while in lobby
There are two types of **player objects** with the NetworkLobbyManager:

LobbyPlayer Object

One for each player
Created when client connects, or player is added
Persists until client disconnects
Holds ready flag and configuration data
Handles commands in the lobby
should use the NetworkLobbyPlayer component
GamePlayer Object

One for each player
Created when game scene is started
Destroyed when re-entering lobby
Handles commands in the game

Properties

Property:	*Function:*
showLobbyGUI	Show the developer OnGUI controls for the lobby.
maxPlayers	The maximum number of players allowed in the lobby.
maxPlayersPerConnection	The maximum number of players allowed to be added for each client connection.
lobbyPlayerPrefab	The **prefab** to create for players when they enter the lobby.
gamePlayerPrefab	The prefab to create for players when the game starts.
lobbyScene	The scene to use for the lobby.
playScene	The scene to use for main game play.

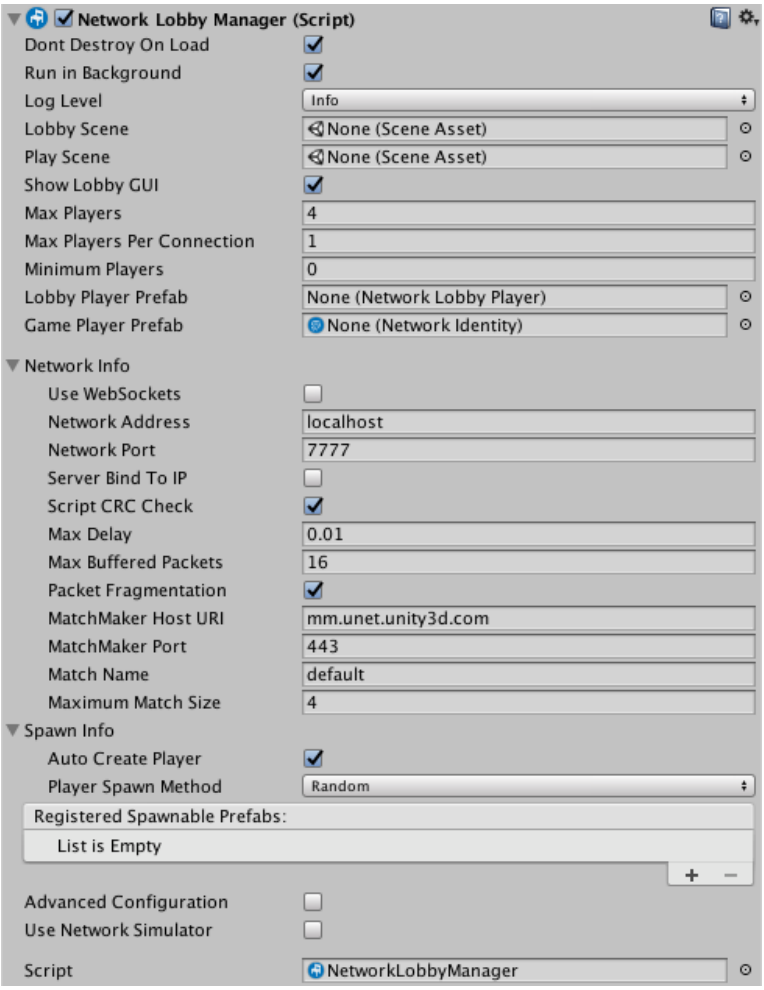
Details

The lobbyPlayerPrefab slot should be filled by an object with the NetworkLobbyPlayer **component** on it.
There is a GUI for the lobby manager. See the multiplayer-lobby **asset** package.

Network Lobby Manager

The Network Lobby Manager is a specialized type of NetworkManager that provides an easy-to-use multiplayer lobby before entering the main play scene of the game.

The Network Lobby Manager has many built-in features that are common to multiplayer games. For instance, it supports setting a maximum player limit, automatically starting the game when all players are ready, and an option to prevent players from joining an in-progress game. The Network Lobby Manager also supports "Couch Multiplayer" where multiple players are using the same client to play together.



The Network Lobby Manager Component

Property:	*Function:*
Show Lobby GUI	Enable this to show the developer GUI controls for the lobby. This is only intended to be used to facilitate developer ease of use. You should create your own **UI** for players to use in your finished game.
Max Players	The maximum number of players allowed in the lobby.
Max Players Per Connection	The maximum number of players allowed to be added for each client connection.
Min Players	The minimum number of players required by the lobby.
Lobby Player Prefab	The prefab to create for players when they enter the lobby.
Game Player Prefab	The prefab to create for players when the game starts.
Lobby Scene	The scene to use for the lobby.
Play Scene	The scene to use for main game play.
Network Info	You can expand this section of the **inspector** to access network-related settings, listed below
Use Web Sockets	When running as a host, enable this setting to make the host listen for WebSocket connections instead of normal transport layer connections, so that **WebGL** clients can connect to it (if you build your game for the WebGL platform). These WebGL instances of your game cannot act as a host (in either peer-hosted or server-only mode). Therefore, for WebGL instances of your multiplayer game to be able to find each other and play together, you must host a server-only instance of your game running in LAN mode, with a publicly reachable IP address, and it must have this option enabled. This checkbox is unticked by default.
Network Address	The network address currently in use. For clients, this is the address of the server that is connected to. For servers, this is the local address. This is set to 'localhost' by default.
Network Port	The network port currently in use. For clients, this is the port of the server connected to. For servers, this is the listen port. This is set to 7777 by default.

Property:	*Function:*
Server Bind To IP	Allows you to tell the server whether to bind to a specific IP address. If this checkbox is not ticked, then there is no specific IP address bound to (IP_ANY). This checkbox is unticked by default. Use this if your server has multiple network addresses (eg, internal LAN, external internet, VPN) and you want to specific the IP address to serve your game on.
Server Bind Address	This field is only visible when the Server Bind To IP checkbox is ticked. Use this to enter the specific IP address that the server should bind to.
Script CRC Check	When this is enabled, Unity checks that the clients and the server are using matching **scripts**. This is useful to make sure outdated versions of your client are not connecting to the latest (updated) version of your server. This checkbox is ticked by default. It does this by performing a (CRC check) [https://en.wikipedia.org/wiki/Cyclic_redundancy_check] between the server and client that ensures the NetworkBehaviour scripts match. This may not be appropriate in some cases, such as when you are intentionally using different Unity projects for the client and server. In most other cases however, you should leave it enabled.
Max Delay	The maximum time in seconds to delay buffered messages. The default of 0.01 seconds means packets are delayed at most by 10 milliseconds. Setting this to zero disables HLAPI connection buffering. This is set to 0.01 by default.
Max Buffered Packets	The maximum number of packets that a NetworkConnection can buffer for each channel. This corresponds to the ChannelOption.MaxPendingBuffers channel option. This is set to 16 by default.
Packet Fragmentation	This allows the `NetworkConnection` instances to fragment packets that are larger than `maxPacketSize`, to up a maximum of 64K. This can cause delays in sending large packets. This checkbox is ticked by default.
MatchMaker Host URI	The host address for the MatchMaker server. By default this points to the global Unity Multiplayer Service at mm.unet.unity3d.com, and usually you should not need to change this. Unity automatically groups players of your game into regional servers around the world, which ensures fast multiplayer response times between players in the same region. This means, for example, that players from Europe, the US, and Asia generally end up playing with other players from their same global region. You can override this value to explicitly control which regional server your game connects to. You might want to do this via scripting if you want to give your players the option of joining a server outside of their global region. For example, if "Player A" (in the US) wanted to connect to a match created via matchmaker by "Player B" (in Europe), they would need to be able to set their desired global region in your game. Therefore you would need to write a UI feature which allows them to select this. See API reference documentation on NetworkMatch.baseUri for more information, and for the regional server URIs.
MatchMaker Port	The host port for the Matchmaker server. By default this points to port 443, and usually you should not need to change this.
Match Name	Define the name of the current match. This is set to "default" by default.
Maximum Match Size	Define the maximum number of players in the current match. This is set to 4 by default.
SpawnInfo	You can expand this section of the inspector to access spawn-related settings, listed below
Player Prefab	Define the default prefab Unity should use to create player **GameObjects** on the server. Unity creates Player **GameObjects** in the default handler for AddPlayer on the server. Implement (OnServerAddPlayer) [https://docs.unity3d.com/ScriptReference/Networking.NetworkManager.OnServerAddPlayer.html] to override this behaviour.
Auto Create Player	Tick this checkbox if you want Unity to automatically create player GameObjects on connect, and when the Scene changes. This checkbox is ticked by default. Note that if you are using the MigrationManager and you do not enable Auto Create Player, you need to call ClientScene.SendReconnectMessage when your client reconnects.
Player Spawn Method	Define how Unity should decide where to spawn new player GameObjects. This is set to Random by default.
Random	Choose Random to spawn players at randomly chosen startPositions.
Round Robin	Choose Round Robin to cycle through startPositions in a set list.
Registered Spawnable Prefabs	Use this list to add prefabs that you want the **Network Manager** to be aware of, so that it can spawn them. You can also add and remove them via scripting.
Advanced Configuration	Tick this checkbox to reveal advanced configuration options in the Network Manager Inspector window.
Max Connections	Define the maximum number of concurrent network connections to support. This is set to 4 by default.

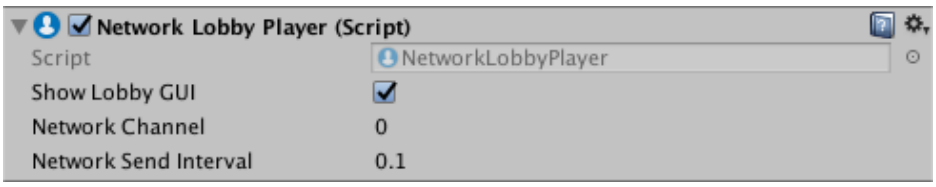
Property:	*Function:*
Qos Channels	A list containing the different communication channels the current Network Manager has, and the Quality Of Service (QoS) setting for each channel. Use this list to add or remove channels, and adjust their QoS setting. You can also configure the channels via scripting. For the descriptions of each QoS option, see QosType.
Timeouts	
Min Update Timeout	Set the minimum time (in milliseconds) the network thread waits between sending network messages. The network thread doesn't send multiplayer network messages immediately. Instead, it check each connection periodically at a fixed rate to see if it has something to send. This is set to 10ms by default. See API reference documentation on MinUpdateTimeout for more information.
Connect Timeout	Define the amount of time (in milliseconds) Unity should wait while trying to connect before attempting the connection again. This is set to 2000ms by default. See API reference documentation on ConnectTimeout for more information.
Disconnect Timeout	The amount of time (in milliseconds) before Unity considers a connection to be disconnected. This is set to 2000ms by default. See API reference documentation on DisconnectTimeout for more information.
Ping Timeout	The amount of time (in milliseconds) between sending pings (also known as "keep-alive" packets). The ping timeout duration should be approximately one-third to one-quarter of the Disconnect Timeout duration, so that Unity doesn't assume that clients are disconnected until the server has failed to receive at least three pings from the client. This is set to 500ms by default. See API reference documentation on ConnectionConfig.PingTimeout for more information.
Global Config	These settings relate to the Reactor. The Reactor is the part of the multiplayer system which receives network packets from the underlying operating system, and passes them into the multiplayer system for processing.
Thread Awake Timeout	The timeout duration in milliseconds, used by the Reactor. How the Reactor uses this value depends on which Reactor Model you select (see below). This is set to 1ms by default.
Reactor Model	Choose which type of reactor to use. The reactor model defines how Unity reads incoming packets. For most games and applications, the default Select reactor is appropriate. If you want to trade a small delay in the processing of network messages for lower CPU usage and improved battery life, use the Fix Rate reactor.
Select Reactor	This model uses the `select()` API which means that the network thread "awakens" (becomes active) as soon as a packet is available. Using this method means your game gets the data as fast as possible. This is the default Reactor Model setting.
Fix Rate Reactor	This model lets the network thread sleep manually for a given amount of time (defined by the value in Thread Awake Timeout) before checking whether there are incoming packets waiting to be processed.
Reactor Max Recv Messages	Set the maximum number of messages stored in the receive queue. This is set to 1024 messages by default.
Reactor Max Sent Messages	Set the maximum number of messages stored in the send queue. This is set to 1024 messages by default.
Use Network Simulator	Tick this checkbox to enable the usage of the network simulator. The network simulator introduces simulated latency and packet loss based on the following settings:
Simulated Average Latency	The amount of delay in milliseconds to simulate.
Simulated Packet Loss	The amount of packet loss to simulate in percent.

Network Lobby Player

Leave feedback

The Network Lobby Player stores per-player state for the Network Lobby Manager while in the lobby. When using this **component**, you need to write a script which allows players to indicate they are ready to begin playing, which sets the readyToBegin property. See documentation on the Network Lobby Manager for full details on how to use this component.

A **GameObject** with a Network Lobby Player component must also have a Network Identity component. When you create a Network Lobby Player component on a GameObject, Unity also creates a Network Identity component on that GameObject if it does not already have one.



The Network Lobby Player component

Property	Function
Show Lobby GUI	Enable this to show the developer GUI for players in the lobby. This **UI** is only intended to be used for ease of development. This is enabled by default.
Network Channel	The network channel used by the Network Lobby Player
Network Send Interval	The rate at which information is sent from the Network Lobby Player to the server. See GetNetworkSendInterval()

Network Manager HUD

Leave feedback

The **Network Manager** HUD is a simple and quick way to provide the basic functions that players of your game need in order to host a networked game, or find and join an existing networked game. It displays a collection of simple **UI** buttons which appear in the Game view when the Editor is in Play mode. It is intended as a helpful short term solution to get you started quickly with making your game. You should replace the **UI** it provides with your own that better suits your game when you are ready.

For a full guide to using the Network Manager HUD **component**, and the controls that are displayed at runtime, see Using the Network Manager HUD.

The Network Manager HUD component in the Inspector window

Property	Function
Show Runtime GUI	Tick this checkbox to show the Network Manager HUD GUI at run time. This allows you to reveal or hide it for quick debugging.
GUI Horizontal Offset	Set the horizontal **pixel** offset of the HUD, measured from the left edge of the screen.
GUI Vertical Offset	Set the vertical pixel offset of the HUD, measured from the top edge of the screen.

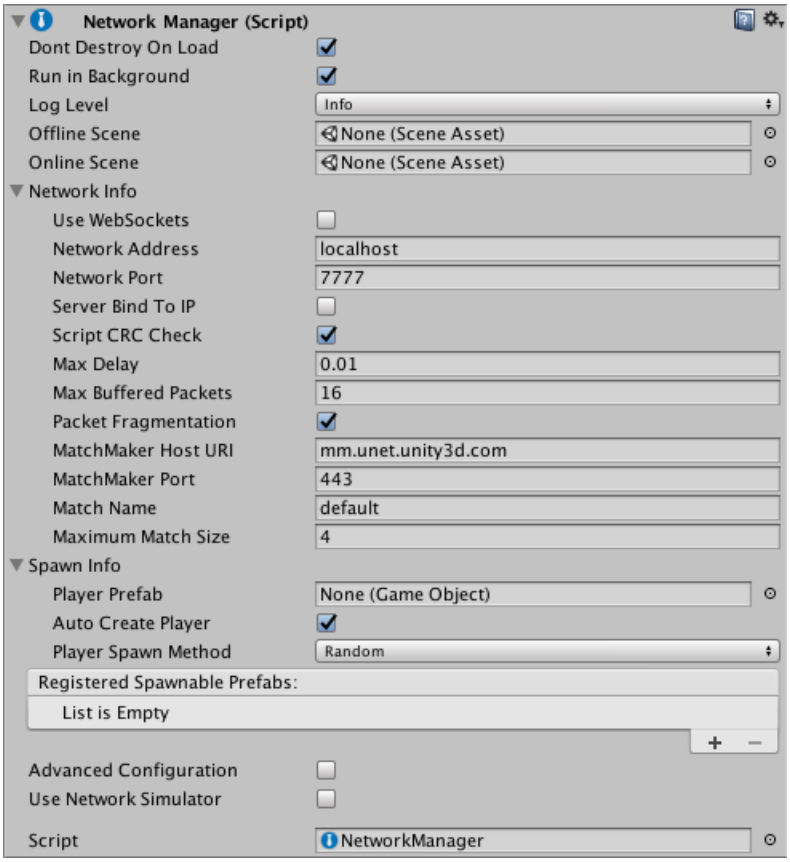
*Note: the **Network Manager HUD** is designed as a temporary aid to development. It allows you to get your multiplayer game running quickly, but you should replace it with your own UI controls when you are ready.*

Network Manager

Leave feedback

The **Network Manager component** allows you to control the state of a networked game. It provides an interface in the Editor for you to configure the network, the **Prefabs** you use for spawning **GameObjects**, and the **Scenes** you use for different game states.

For more details on implementing the Network Manager in your game, see documentation on Using the Network Manager.



The Network Manager component in the Inspector window

Property	Function
Dont Destroy On Load	Use this property to control whether or not Unity should destroy the GameObject with the Network Manager when the Scene changes. Tick this checkbox to ensure Unity does not destroy your Network Manager GameObject when the Scene changes in your game. Untick the checkbox if you want Unity to destroy the GameObject when the Scene it exists in is no longer the active Scene. This is useful if you want to manage multiple, separate Network Manager GameObjects in each of your Scenes. This checkbox is ticked by default.
Run In Background	Use this property to control whether the networked game runs when the window it is running in is not focused. Tick the checkbox if you want it to run; untick it if you want the game to stop running when the window is not focused. This checkbox is ticked by default. You need to enable this property if you want to run multiple instances of a program on the same machine, such as when testing using localhost. You should disable it when deploying to mobile platforms. When enabled, it sets Application.runInBackground to true when the Network Manager starts up. You can also set this property from the Unity menu: Edit > Project Settings > Player Settings > Resolution & Presentation.
Log Level	Use this property to control the amount of information Unity outputs to the **console window**. A low level results in more information; a high level results in less information. Each level includes message from all the levels higher than itself (for example, if you select “Warn”, the console also prints outputs all “Error” and “Fatal” log messages). The drop-down lists the levels from low to high. This property is set to Info by default. You can set Log Level to **Set in Scripting** to prevent the Network Manager from setting the log level at all. This means you can control the level from your own **scripts** instead.
Offline Scene	If you assign a Scene to this field, the Network Manager automatically switches to the specified Scene when a network session stops - for example, when the client disconnects, or when the server shuts down.

Property	Function
Online Scene	If you assign a Scene to this field, the Network Manager automatically switches to the specified Scene when a network session starts - for example, when the client connects to a server, or when the server starts listening for connections.
Network Info	You can expand this section of the **inspector** to access network-related settings, listed below
Use Web Sockets	When running as a host, enable this setting to make the host listen for WebSocket connections instead of normal transport layer connections, so that **WebGL** clients can connect to it (if you build your game for the WebGL platform). These WebGL instances of your game cannot act as a host (in either peer-hosted or server-only mode). Therefore, for WebGL instances of your multiplayer game to be able to find each other and play together, you must host a server-only instance of your game running in LAN mode, with a publicly reachable IP address, and it must have this option enabled. This checkbox is unticked by default.
Network Address	The network address currently in use. For clients, this is the address of the server that is connected to. For servers, this is the local address. This is set to 'localhost' by default.
Network Port	The network port currently in use. For clients, this is the port of the server connected to. For servers, this is the listen port. This is set to 7777 by default.
Server Bind To IP	Allows you to tell the server whether to bind to a specific IP address. If this checkbox is not ticked, then there is no specific IP address bound to (IP_ANY). This checkbox is unticked by default. Use this if your server has multiple network addresses (eg, internal LAN, external internet, VPN) and you want to specific the IP address to serve your game on.
Server Bind Address	This field is only visible when the Server Bind To IP checkbox is ticked. Use this to enter the specific IP address that the server should bind to.
Script CRC Check	When this is enabled, Unity checks that the clients and the server are using matching scripts. This is useful to make sure outdated versions of your client are not connecting to the latest (updated) version of your server. This checkbox is ticked by default. It does this by performing a (CRC check) [https://en.wikipedia.org/wiki/Cyclic_redundancy_check] between the server and client that ensures the NetworkBehaviour scripts match. This may not be appropriate in some cases, such as when you are intentionally using different Unity projects for the client and server. In most other cases however, you should leave it enabled.
Max Delay	The maximum time in seconds to delay buffered messages. The default of 0.01 seconds means packets are delayed at most by 10 milliseconds. Setting this to zero disables HLAPI connection buffering. This is set to 0.01 by default.
Max Buffered Packets	The maximum number of packets that a NetworkConnection can buffer for each channel. This corresponds to the ChannelOption.MaxPendingBuffers channel option. This is set to 16 by default.
Packet Fragmentation	This allows the `NetworkConnection` instances to fragment packets that are larger than `maxPacketSize`, to up a maximum of 64K. This can cause delays in sending large packets. This checkbox is ticked by default.
MatchMaker Host URI	The host address for the MatchMaker server. By default this points to the global Unity Multiplayer Service at mm.unet.unity3d.com, and usually you should not need to change this. Unity automatically groups players of your game into regional servers around the world, which ensures fast multiplayer response times between players in the same region. This means, for example, that players from Europe, the US, and Asia generally end up playing with other players from their same global region. You can override this value to explicitly control which regional server your game connects to. You might want to do this via scripting if you want to give your players the option of joining a server outside of their global region. For example, if "Player A" (in the US) wanted to connect to a match created via matchmaker by "Player B" (in Europe), they would need to be able to set their desired global region in your game. Therefore you would need to write a **UI** feature which allows them to select this. See API reference documentation on NetworkMatch.baseUri for more information, and for the regional server URIs.
MatchMaker Port	The host port for the Matchmaker server. By default this points to port 443, and usually you should not need to change this.
Match Name	Define the name of the current match. This is set to "default" by default.
Maximum Match Size	Define the maximum number of players in the current match. This is set to 4 by default.
SpawnInfo	You can expand this section of the inspector to access spawn-related settings, listed below
Player Prefab	Define the default prefab Unity should use to create player GameObjects on the server. Unity creates Player GameObjects in the default handler for AddPlayer on the server. Implement (OnServerAddPlayer) [https://docs.unity3d.com/ScriptReference/Networking.NetworkManager.OnServerAddPlayer.html] to override this behaviour.

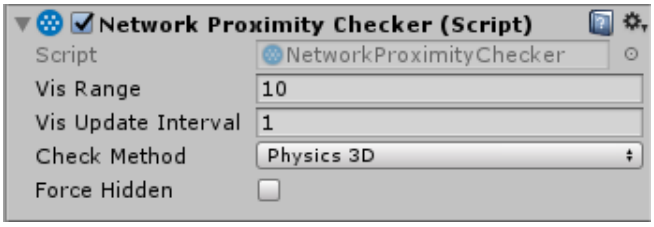
Property	Function
Auto Create Player	Tick this checkbox if you want Unity to automatically create player GameObjects on connect, and when the Scene changes. This checkbox is ticked by default. Note that if you are using the MigrationManager and you do not enable Auto Create Player, you need to call ClientScene.SendReconnectMessage when your client reconnects.
Player Spawn Method	Define how Unity should decide where to spawn new player GameObjects. This is set to Random by default.
Random	Choose Random to spawn players at randomly chosen startPositions.
Round Robin	Choose Round Robin to cycle through startPositions in a set list.
Registered Spawnable Prefabs	Use this list to add prefabs that you want the Network Manager to be aware of, so that it can spawn them. You can also add and remove them via scripting.
Advanced Configuration	Tick this checkbox to reveal advanced configuration options in the Network Manager Inspector window.
Max Connections	Define the maximum number of concurrent network connections to support. This is set to 4 by default.
Qos Channels	A list containing the different communication channels the current Network Manager has, and the Quality Of Service (QoS) setting for each channel. Use this list to add or remove channels, and adjust their QoS setting. You can also configure the channels via scripting. For the descriptions of each QoS option, see QosType.
Timeouts	
Min Update Timeout	Set the minimum time (in milliseconds) the network thread waits between sending network messages. The network thread doesn't send multiplayer network messages immediately. Instead, it check each connection periodically at a fixed rate to see if it has something to send. This is set to 10ms by default. See API reference documentation on MinUpdateTimeout for more information.
Connect Timeout	Define the amount of time (in milliseconds) Unity should wait while trying to connect before attempting the connection again. This is set to 2000ms by default. See API reference documentation on ConnectTimeout for more information.
Disconnect Timeout	The amount of time (in milliseconds) before Unity considers a connection to be disconnected. This is set to 2000ms by default. See API reference documentation on DisconnectTimeout for more information.
Ping Timeout	The amount of time (in milliseconds) between sending pings (also known as "keep-alive" packets). The ping timeout duration should be approximately one-third to one-quarter of the Disconnect Timeout duration, so that Unity doesn't assume that clients are disconnected until the server has failed to receive at least three pings from the client. This is set to 500ms by default. See API reference documentation on ConnectionConfig.PingTimeout for more information.
Global Config	These settings relate to the Reactor. The Reactor is the part of the multiplayer system which receives network packets from the underlying operating system, and passes them into the multiplayer system for processing.
Thread Awake Timeout	The timeout duration in milliseconds, used by the Reactor. How the Reactor uses this value depends on which Reactor Model you select (see below). This is set to 1ms by default.
Reactor Model	Choose which type of reactor to use. The reactor model defines how Unity reads incoming packets. For most games and applications, the default Select reactor is appropriate. If you want to trade a small delay in the processing of network messages for lower CPU usage and improved battery life, use the Fix Rate reactor.
Select Reactor	This model uses the `select()` API which means that the network thread "awakens" (becomes active) as soon as a packet is available. Using this method means your game gets the data as fast as possible. This is the default Reactor Model setting.
Fix Rate Reactor	This model lets the network thread sleep manually for a given amount of time (defined by the value in Thread Awake Timeout) before checking whether there are incoming packets waiting to be processed.
Reactor Max Recv Messages	Set the maximum number of messages stored in the receive queue. This is set to 1024 messages by default.
Reactor Max Sent Messages	Set the maximum number of messages stored in the send queue. This is set to 1024 messages by default.

Property	Function
Use Network Simulator	Tick this checkbox to enable the usage of the network simulator. The network simulator introduces simulated latency and packet loss based on the following settings:
Simulated Average Latency	The amount of delay in milliseconds to simulate.
Simulated Packet Loss	The amount of packet loss to simulate in percent.

Network Proximity Checker

Leave feedback

The Network Proximity Checker **component** controls the visibility of **GameObjects** for network clients, based on proximity to players.



The Network Proximity Checker component

Property	Function
Vis Range	Define the range that the GameObject should be visible in.
Vis Update Interval	Define how often (in seconds) the GameObject should check for players entering its visible range.
Check Method	Define which type of physics (2D or 3D) to use for proximity checking.
Force Hidden	Tick this checkbox to hide this object from all players.

With the Network Proximity Checker, a game running on a client doesn't have information about GameObjects that are not visible. This has two main benefits: it reduces the amount of data sent across the network, and it makes your game more secure against hacking.

This component relies on physics to calculate visibility, so the GameObject must also have a **collider** component on it.

A GameObject with a Network Proximity Checker component must also have a Network Identity component. When you create a Network Proximity Checker component on a GameObject, Unity also creates a Network Identity component on that GameObject if it does not already have one.

NetworkStartPosition

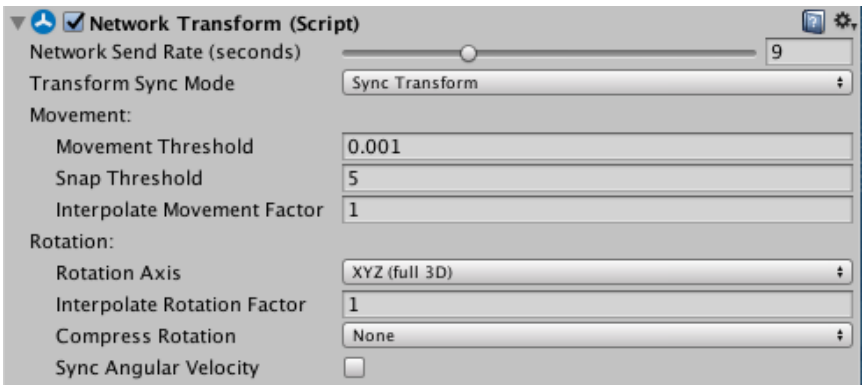
Leave feedback

NetworkStartPosition is used by the NetworkManager when creating player objects. The position and rotation of the NetworkStartPosition are used to place the newly created **player object**.

Network Transform

Leave feedback

The Network Transform **component** synchronizes the movement and rotation of **GameObjects** across the network. Note that the network Transform **component** only synchronizes spawned networked **GameObjects**.



The Network Transform component

Property	Function
Network Send Rate (seconds)	Set the number of network updates per second. You can set this slider to 0 for GameObjects that do not need to update after being created, like non-interactive effects generated by a player (for example, a dust cloud left behind that the player cannot interact with).
Transform Sync Mode	Select what type of synchronization should occur on this GameObject.
Sync None	Don't synchronize.
Sync Transform	Use the GameObject's Transform for synchronization. Use this if the physics system does not control this GameObject (that is, if you are moving it via scripting or animation). This is the default option.
Sync Rigidbody 2D	Use the Rigidbody2D component for synchronization. Use this if the 2D physics system controls this GameObject.
Sync Rigidbody 3D	Use the Rigidbody component for synchronization. Use this if the 3D physics system controls this GameObject.
Sync Character Controller	Use the **Character Controller** component for synchronization. Only select this if you're using a **Character Controller**.
Movement:	
Movement Threshold	Set the distance that a GameObject can move without sending a movement synchronization update.
Snap Threshold	Set the threshold at which, if a movement update puts a GameObject further from its current position than this, the GameObject snaps to the position instead of moving smoothly.
Interpolate Movement Factor	Use this to enable and control **interpolation** of the synchronized movement. The larger this number is, the faster the GameObject interpolates to the **target position**. If this is set to 0, the GameObject snaps to the new position.
Rotation:	
Rotation Axis	Define which rotation axis or axes should synchronize. This is set to XYZ (full 3D) by default.
Interpolate Rotation Factor	Use this to enable and control interpolation of the synchronized rotation. The larger this number is, the faster the GameObject interpolates to the target rotation. If this is set to 0, the GameObject snaps to the new rotation.
Compress Rotation	If you compress rotation data, the amount of data sent is lower, and the accuracy of the rotation synchronization is lower.
None	Choose this to apply no **compression** to the rotation synchronization. This is the default option.

Property	Function
Low	Choose this to apply a low amount of compression to the rotation synchronization. This option lessens the amount of information sent for rotation data.
High	Choose this to apply a high amount of compression to the rotation synchronization. This option sends the least amount of information possible for rotation data.
Sync Angular Velocity	Tick this checkbox to synchronize the angular velocity of the attached Rigidbody component.

This component takes authority into account, so local player GameObjects (which have local authority) synchronize their position from the client to server, then out to other clients. Other GameObjects (with server authority) synchronize their position from the server to clients.

A GameObject with a Network **Transform component** must also have a Network Identity component. When you create a Network Transform component on a GameObject, Unity also creates a Network Identity component on that GameObject if it does not already have one.

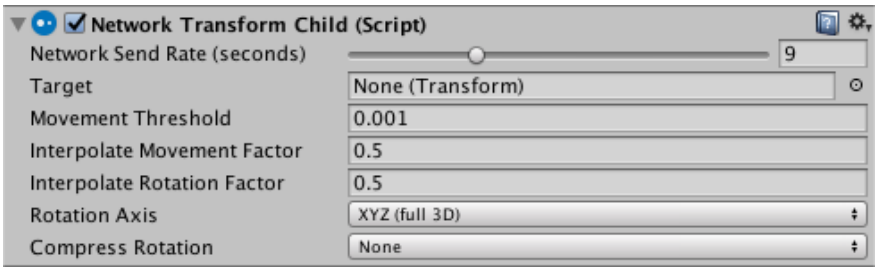
Note that the Network Transform Visualizer component is a useful tool for debugging the Network Transform component.

Network Transform Child

Leave feedback

The Network Transform Child **component** synchronizes the position and rotation of the child **GameObject** of a **GameObject** with a Network Transform component. You should use this component in situations where you need to synchronize an independently-moving child object of a Networked GameObject.

To use the Network Transform Child component, attach it to the same parent GameObject as the Network Transform, and use the **Target** field to define which child GameObject to apply the component settings to. You can have multiple Network Transform Child components on one parent GameObject.



The Network Transform Child component

Property	Function	
Network Send Rate (seconds)	**|Set the number of network updates per second. You can set this slider to 0 for GameObjects that do not need to update after being created, like non-interactive effects generated by a player (for example, a dust cloud left behind that the player cannot interact with).| |**Target**|**	Child transform to be synchronized. (Remember, this component goes on the *parent*, not the child - so you specify the child object using this field).
Movement Threshold	**|Set the distance that a GameObject can move without sending a movement synchronization update.| |**Interpolate Movement Factor**|**	Use this to enable and control **interpolation** of the synchronized movement. The larger this number is, the faster the GameObject interpolates to the **target position**. If this is set to 0, the GameObject snaps to the new position.
Interpolate Rotation Factor	**|Use this to enable and control interpolation of the synchronized rotation. The larger this number is, the faster the GameObject interpolates to the target rotation. If this is set to 0, the GameObject snaps to the new rotation.| |**Rotation Axis**|**	Define which rotation axis or axes should synchronize. This is set to XYZ (full 3D) by default.
Compress Rotation	**|If you compress rotation data, the amount of data sent is lower, and the accuracy of the rotation synchronization is lower.| | None|**	Choose this to apply no **compression** to the rotation synchronization. This is the default option.
Low	**|Choose this to apply a low amount of compression to the rotation synchronization. This option lessens the amount of information sent for rotation data.| | High|**	Choose this to apply a high amount of compression to the rotation synchronization. This option sends the least amount of information possible for rotation data.

This component does not use physics; it synchronizes the position and rotation of the child GameObject, and interpolates towards updated values. Use **Interpolate Movement Factor** and **Interpolate Rotation Factor** to customize the rate of interpolation.

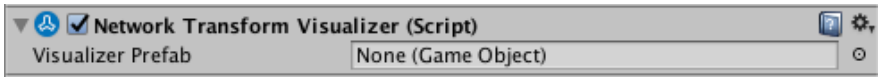
A GameObject with a Network Transform Child component must also have a Network Identity component. When you create a Network Transform Child component on a GameObject, Unity also creates a Network Identity component on that GameObject if it does not already have one.

Network Transform Visualizer

Leave feedback

The Network Transform Visualizer is a utility **component** that allows you to visualize the **interpolation** of **GameObjects** that use the Network Transform component. To use it, add it to a GameObject that already has a Network **Transform component**, and assign a **Prefab** in the **Inspector**. The **Prefab** can be anything you choose, it will be used as a visual representation of the incoming transform data for GameObject.

GameObjects with local authority (such as the local player) are not interpolated, and therefore won't show a visualizer GameObject. The visualizer will only show other Networked GameObjects controlled by other computers on the network (such as other players).



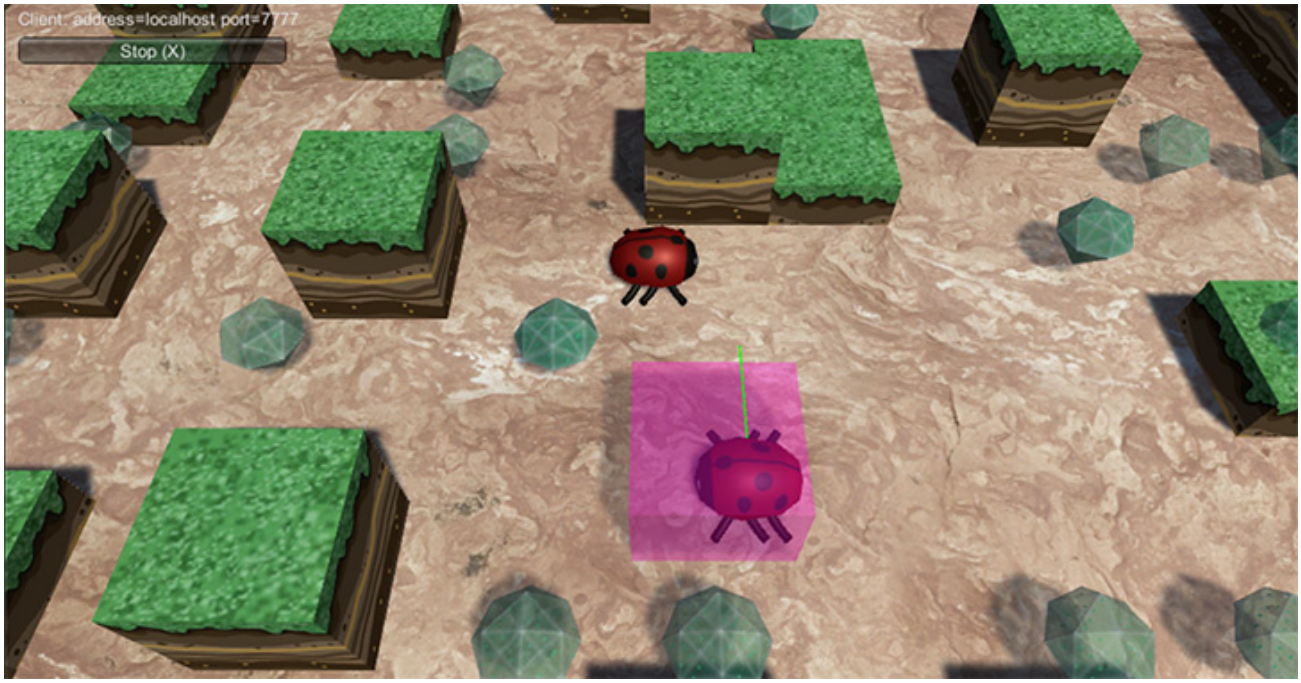
The Network Transform Visualizer component

The Network Transform Visualizer component in the Inspector window

Property	Function
Visualizer Prefab	Define the Prefab used to visualize the **target position** of the network transform.

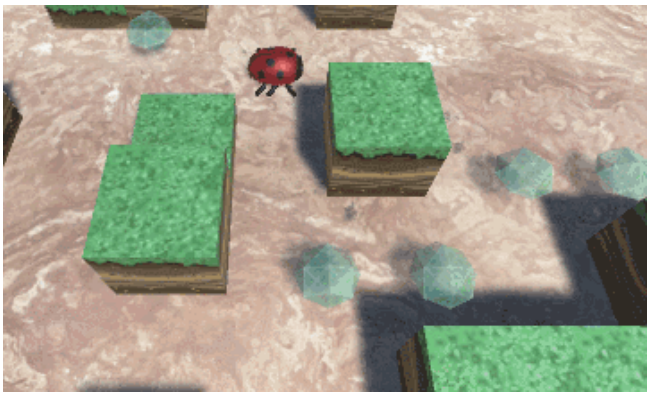
When the game is playing, the Prefab is instantiated as the "visualizer" GameObject. When the Network Transform GameObject moves, the visualizer GameObject is displayed at the target position of the Network Transform.

You can choose whatever you like to be the visualizer prefab. In the example below, a semi-transparent magenta cube is used.



In this image the Network Transform Visualizer is showing the incoming transform data for a remote player in the game, represented by the magenta cube.

It usually appears to be moving a little ahead of - but less smoothly than - the Network Transform GameObject. This is because it is showing you the raw positional data coming in directly from the network, rather than using interpolation to smoothly reach each new target position.



This animation shows that the incoming network tranform data is slightly ahead but less smooth than the interpolated position of the networked GameObject

A GameObject with a Network Transform Visualizer component must also have a Network Identity component. When you create a Network Transform Visualizer component on a GameObject, Unity also creates a Network Transform component and a Network Identity component on that GameObject if it does not already have one.

Note: Make sure the prefab you choose to use as your visualization GameObject does not have a **collider** attached, or anything else that could affect the gameplay of your game!

Multiplayer Classes Reference

Leave feedback

You can create **scripts** which inherit from these classes to customise the way Unity's **networking** works.

The NetworkBehaviour class works with **GameObjects** that have a Network Identity **component**. These scripts can perform high-level API functions such as Commands, ClientRPCs, SyncEvents and SyncVars.
The NetworkClient class manages a network connection from a client to a server, and can send and receive messages between the client and the server.
The NetworkConnection encapsulates a network connection. (NetworkClient)[class-NetworkClient] objects have a `NetworkConnection`, and NetworkServers have multiple connections - one from each client.
NetworkConnections have the ability to send byte arrays, or serialized objects as network messages.
The NetworkServer manages connections from multiple clients, and provides game-related functionality such as spawning, local clients, and player manager.
The NetworkServerSimple is a basic server class with no game-related functionality. While the NetworkServer class handles game-like things such as spawning, local clients, and player manager, and has a static interface, the NetworkServerSimple class is a pure network server with no game related functionality. It also has no static interface or singleton, so more than one instance can exist in a process at a time.

NetworkBehaviour

Leave feedback

NetworkBehaviour **scripts** work with **GameObjects** that have a Network Identity **component**. These scripts can perform high-level API functions such as Commands, ClientRPCs, SyncEvents and SyncVars.

With the server-authoritative system of the Unity Network System, the server must use the NetworkServer.Spawn function to spawn GameObjects with Network Identity components. Spawning them this way assigns them a NetworkInstanceId and creates them on clients connected to the server.

Note: This is not a component that you can add to a GameObject directly. Instead, you must create a script which inherits from `NetworkBehaviour` (instead of the default `MonoBehaviour`), then you can add your script as a component to a GameObject.

Properties

Property	Description
`isLocalPlayer`	Returns true if this GameObject is the one that represents the player on the local client.
`isServer`	Returns true if this GameObject is running on the server, and has been spawned.
`isClient`	Returns true if this GameObject is on the client and has been spawned by the server.
`hasAuthority`	Returns true if this GameObject is the authoritative version of the GameObject, meaning it is the source for changes to be synchronized. For most GameObjects, this returns true on the server. However, if the localPlayerAuthority value on the **NetworkIdentity** is true, the authority rests with that player's client, and this value is true on that client instead of on the server.
`netId`	The unique network ID of this GameObject. The server assigns this at runtime. It is unique for all GameObjects in that network session.
`playerControllerId`	The ID of the player associated with this NetworkBehaviour script. This is only valid if the object is a local player.
`connectionToServer`	The NetworkConnection associated with the Network Identity component attached to this GameObject. This is only valid for **player objects** on the client.
`connectionToClient`	The NetworkConnection associated with the Network Identity component attached to this GameObject. This is only valid for player GameObjects on the server.
`localPlayerAuthority`	This value is set on the Network Identity component and is accessible from the NetworkBehaviour script for convenient access in scripts.

NetworkBehaviour scripts have the following features:

Synchronized variables

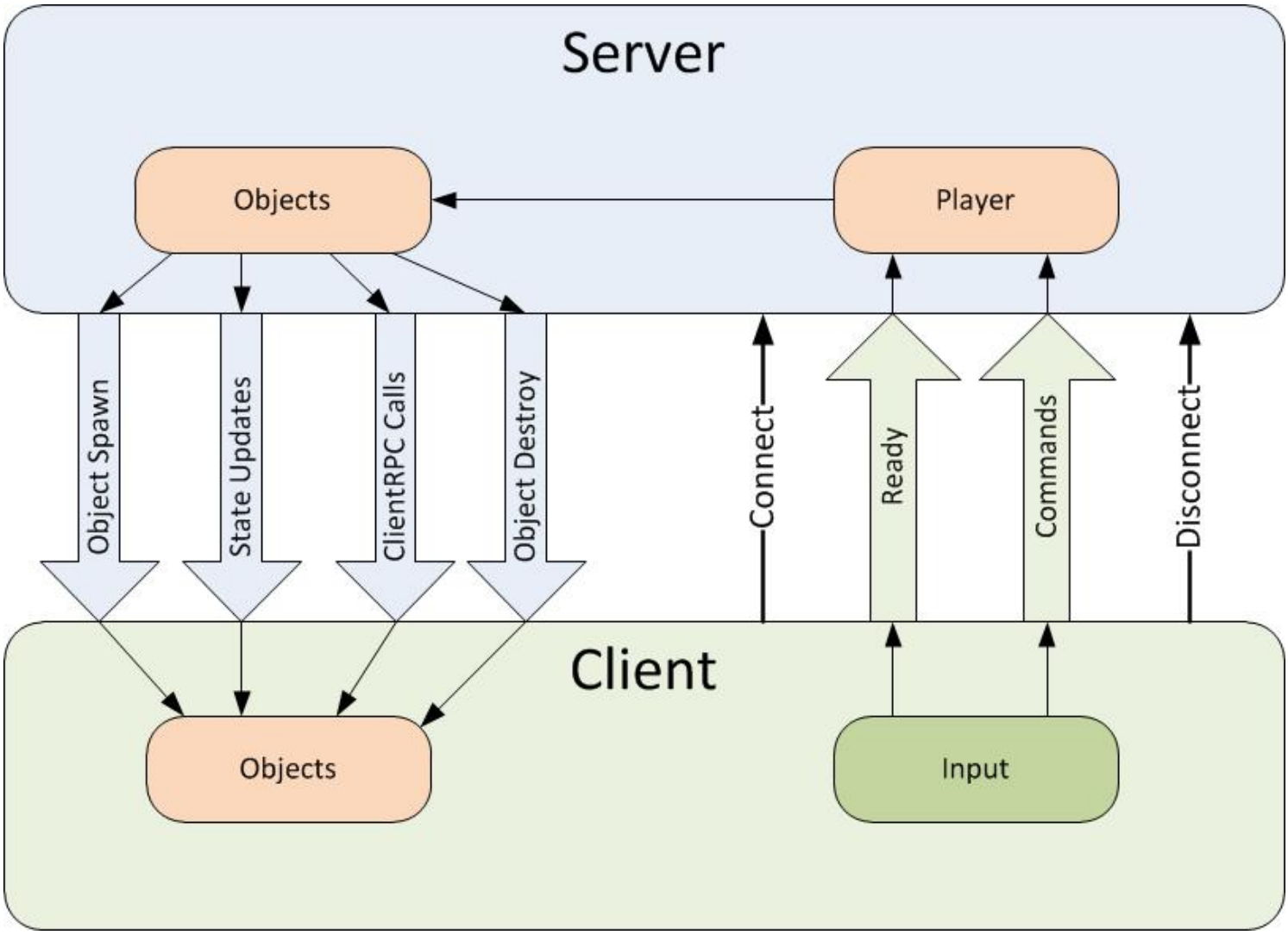
Network callbacks

Server and client functions

Sending commands

Client RPC calls

Networked events



Synchronized variables

You can synchronize member variables of NetworkBehaviour scripts from the server to clients. The server is authoritative in this system, so synchronization only takes place in the direction of server to client.

Use the SyncVar attribute to tag member variables as synchronized. Synchronized variables can be any basic type (bool, byte, sbyte, char, decimal, double, float, int, uint, long, ulong, short, ushort, string), but not classes, lists, or other collections.

```
public class SpaceShip : NetworkBehaviour
{
    [SyncVar]
    public int health;

    [SyncVar]
```

```
    public string playerName;
}
```

When the value of a `SyncVar` changes on the server, the server automatically sends the new value to all ready clients in the game, and updates the corresponding `SyncVar` values on those clients. When GameObjects spawn, they are created on the client with the latest state of all `SyncVar` attributes from the server.

Note: To make a request from a client to the server, you need to use **commands**, not synchronized variables. See documentation on Sending commands for more information.

Network callbacks

There are built-in callback functions which are invoked on NetworkBehaviour scripts for various network events. These are virtual functions on the base class, so you can override them in your own code like this:

```
public class SpaceShip : NetworkBehaviour
{
    public override void OnStartServer()
    {
        // disable client stuff
    }

    public override void OnStartClient()
    {
        // register client events, enable effects
    }
}
```

The built-in callbacks are:

OnStartServer - called when a GameObject spawns on the server, or when the server is started for GameObjects in the Scene

OnStartClient - called when the GameObject spawns on the client, or when the client connects to a server for GameObjects in the Scene

OnSerialize - called to gather state to send from the server to clients

OnDeSerialize - called to apply state to GameObjects on clients

OnNetworkDestroy - called on clients when the server destroys the GameObject

OnStartLocalPlayer - called on clients for player GameObjects on the local client (only)

OnRebuildObservers - called on the server when the set of observers for a GameObjects is rebuilt

OnSetLocalVisibility - called on the client and/or server when the visibility of a GameObject changes for the local client

OnCheckObserver - called on the server to check visibility state for a new client

Note that in a peer-hosted setup, when one of the clients is acting as both host and client, both `OnStartServer` and `OnStartClient` are called on the same GameObject. Both these functions are useful for actions that are specific to either the client or server, such as suppressing effects on a server, or setting up client-side events.

Server and Client functions

You can tag member functions in NetworkBehaviour scripts with custom attributes to designate them as server-only or client-only functions. For example:

```
using UnityEngine;
using UnityEngine.Networking;

public class SimpleSpaceShip : NetworkBehaviour
{
    int health;

    [Server]
    public void TakeDamage( int amount)
    {
        // will only work on server
        health -= amount;
    }

    [ServerCallback]
    void Update()
    {
        // engine invoked callback - will only run on server
    }


    [Client]
    void ShowExplosion()
    {
        // will only run on client
    }

    [ClientCallback]
    void Update()
    {
        // engine invoked callback - will only run on client
```

```
    }
}
```

`[Server]` and `[ServerCallback]` return immediately if the client is not active. Likewise, `[Client]` and `[ClientCallback]` return immediately if the server is not active.

The `[Server]` and `[Client]` attributes are for your own custom callback functions. They do not generate compile time errors, but they do emit a warning log message if called in the wrong scope.

The `[ServerCallback]` and `[ClientCallback]` attributes are for built-in callback functions that are called automatically by Unity. These attributes do not cause a warning to be generated.

For more information, see API reference documentation on the attributes discussed:

ClientAttribute

ClientCallbackAttribute

ServerAttribute

ServerCallbackAttribute

Sending commands

To execute code on the server, you must use **commands**. The high-level API is a server-authoritative system, so commands are the only way for a client to trigger some code on the server.

Only player GameObjects can send commands.

When client player GameObject sends a command, that command runs on the corresponding player GameObject on the server. This routing happens automatically, so it is impossible for a client to send a command for a different player.

To define a **command** in your code, you must write a function which has:

A name that begins with `Cmd`

The `[Command]` attribute

For example:

```
using UnityEngine;
using UnityEngine.__Networking__;

public class SpaceShip : NetworkBehaviour
{
    bool alive;
```

```
    float thrusting;
    int spin;

    [ClientCallback]
    void Update()
    {

        // This code executes on the client, gathering input
        int spin = 0;
        if (Input.GetKey(KeyCode.LeftArrow))
        {
            spin += 1;
        }
        if (Input.GetKey(KeyCode.RightArrow))
        {
            spin -= 1;
        }


        // This line triggers the code to run on the server
        CmdThrust(Input.GetAxis("Vertical"), spin);
    }

    [Command]
    public void CmdThrust(float thrusting, int spin)
    {

        // This code executes on the server after Update() is

        // called from below.
        if (!alive)
        {
            this.thrusting = 0;
            this.spin = 0;
            return;
        }

        this.thrusting = thrusting;
        this.spin = spin;
    }

}
```

Commands are called just by invoking the function normally on the client. Instead of the command function running on the client, it is automatically invoked on the corresponding player GameObject on the server.

Commands are type-safe, have built-in security and routing to the player, and use an efficient serialization mechanism for the arguments to make calling them fast.

Client RPC calls

Client RPC calls are a way for server GameObjects to make things happen on client GameObjects.

Client RPC calls are not restricted to player GameObjects, and may be called on any GameObject with a Network Identity component.

To define a **client RPC call** in your code, you must write a function which:

Has a name that begins with `Rpc`

Has the `[ClientRPC]` attribute

For example:

```
using UnityEngine;
using UnityEngine.Networking;

public class SpaceShipRpc : NetworkBehaviour
{
    [ServerCallback]
    void Update()
    {

        // This is code run on the server
        int value = UnityEngine.Random.Range(0,100);
        if (value < 10)
        {
            // This invoke the RpcDoOnClient function on all clients
            RpcDoOnClient(value);
        }
    }

    [ClientRpc]
    public void RpcDoOnClient(int foo)
    {

        // This code will run on all clients
        Debug.Log("OnClient " + foo);
    }

}
```

Networked events

Networked events are like **Client RPC** calls, but instead of calling a function on the GameObject, they trigger Events instead.

This allows you to write scripts which can register for a callback when an event is triggered.

To define a **Networked event** in your code, you must write a function which both:

Has a name that begins with `Event`

Has the `[SyncEvent]` attribute

You can use events to build powerful networked game systems that can be extended by other scripts. This example shows how an effect script on the client can respond to events generated by a combat script on the server.

SyncEvent is the base class that **Commands** and **ClientRPC** calls are derived from. You can use the SyncEvent attribute on your own functions to make your own event-driven networked gameplay code. Using SyncEvent, you can extend Unity's Multiplayer features to better fit your own programming patterns. For example:

```
using UnityEngine;
using UnityEngine.Networking;

// Server script
public class MyCombat : NetworkBehaviour
{
    public delegate void TakeDamageDelegate(int amount);
    public delegate void DieDelegate();
    public delegate void RespawnDelegate();

    float deathTimer;
    bool alive;
    int health;

    [SyncEvent(channel=1)]
    public event TakeDamageDelegate EventTakeDamage;

    [SyncEvent]
    public event DieDelegate EventDie;

    [SyncEvent]
    public event RespawnDelegate EventRespawn;
```

```
[ServerCallback]
void Update()
{

    // Check if it is time to Respawn
    if (!alive)
    {
        if (Time.time > deathTimer)
        {
            Respawn();
        }
        return;
    }
}

[Server]
void Respawn()
{
    alive = true;


    // send respawn event to all clients from the Server
    EventRespawn();
}


[Server]
void EventTakeDamage(int amount)
{
    if (!alive)
        return;

    if (health > amount) {
        health -= amount;
    }
    else
    {
        health = 0;
        alive = false;


        // send die event to all clients
        EventDie();
        deathTimer = Time.time + 5.0f;
    }
```

```
        }
}
```

NetworkClient

Leave feedback

`NetworkClient` is a high-level API class that manages a network connection from a client to a server, and can send and receive messages between the client and the server. The `NetworkClient` class also helps to manage spawned network GameObjects, and routing of RPC message and network events.

See the NetworkClient script reference for more information.

Properties

Property:	*Function:*
serverIP	The IP address of the server that this client is connected to.
serverPort	The port of the server that this client is connected to.
connection	The NetworkConnection GameObject this `NetworkClient` instance is using.
handlers	The set of registered message handler functions.
numChannels	The number of configured NetworkTransport QoS channels.
isConnected	True if the client is connected to a server.
allClients	List of active NetworkClients (static).
active	True if any NetworkClients are active (static).

NetworkConnection

Leave feedback

NetworkConnection is a high-level API class that encapsulates a network connection. (NetworkClient)[class-NetworkClient] objects have a `NetworkConnection`, and NetworkServers have multiple connections - one from each client. NetworkConnections have the ability to send byte arrays, or serialized objects as network messages.

Properties

Property:	*Function:*
hostId	The NetworkTransport hostId for this connection.
connectionId	The `NetworkTransport` connectionId for this connection.
isReady	Flag to control whether state updates are sent to this connection
lastMessageTime	The last time that a message was received on this connection.
address	The IP address of the end-point that this connection is connected to.
playerControllers	The set of players that have been added with AddPlayer().
clientOwnedObjects	The set of objects that this connection has authority over.

The NetworkConnection class has virtual functions that are called when data is sent to the transport layer or recieved from the transport layer. These functions allow specialized versions of NetworkConnection to inspect or modify this data, or even route it to different sources. These function are show below, including the default behaviour:

```
public virtual void TransportRecieve(byte[] bytes, int numBytes, int channelId)
{
    HandleBytes(bytes, numBytes, channelId);
}

public virtual bool TransportSend(byte[] bytes, int numBytes, int channelId, out
{
    return NetworkTransport.Send(hostId, connectionId, channelId, bytes, numByte
}
```

An example use of these function is to log the contents of incoming and outgoing packets. Below is an example of a DebugConnection class that is derived from NetworkConnection that logs the first 50 bytes of packets to the console. To use a class like this call the SetNetworkConnectionClass() function on a NetworkClient or NetworkServer.

```
class DebugConnection : NetworkConnection
{
    public override void TransportRecieve(byte[] bytes, int numBytes, int channe
    {
        StringBuilder msg = new StringBuilder();
        for (int i = 0; i < numBytes; i++)
```

```
        {
            var s = String.Format("{0:X2}", bytes[i]);
            msg.Append(s);
            if (i > 50) break;
        }
        UnityEngine.Debug.Log("TransportRecieve h:" + hostId + " con:" + connect

        HandleBytes(bytes, numBytes, channelId);
    }

    public override bool TransportSend(byte[] bytes, int numBytes, int channelId
    {
        StringBuilder msg = new StringBuilder();
        for (int i = 0; i < numBytes; i++)
        {
            var s = String.Format("{0:X2}", bytes[i]);
            msg.Append(s);
            if (i > 50) break;
        }
        UnityEngine.Debug.Log("TransportSend    h:" + hostId + " con:" + connect

        return NetworkTransport.Send(hostId, connectionId, channelId, bytes, num
    }
}
```

NetworkServer

Leave feedback

NetworkServer is a High-Level-API class that manages connections from multiple clients.

Properties

Property:	*Function:*
active	Checks if the server has been started.
connections	A list of all the current connections from clients.
dontListen	If you enable this, the server will not listen for incoming connections on the regular network port.
handlers	Dictionary of the message handlers registered with the server.
hostTopology	The host topology that the server is using.
listenPort	The port that the server is listening on.
localClientActive	True if a local client is currently active on the server.
localConnections	A list of local connections on the server.
maxDelay	The maximum delay before sending packets on connections.
networkConnectionClass	The class to be used when creating new network connections.
numChannels	The number of channels the network is configure with.
objects	This is a dictionary of networked objects that have been spawned on the server.
serverHostId	The transport layer hostId used by this server.
useWebSockets	This makes the server listen for WebSockets connections instead of normal transport layer connections.

NetworkServerSimple

Leave feedback

NetworkServerSimple is a High Level API (HLAPI) class that manages connections from multiple clients. While the NetworkServer class handles game-like things such as spawning, local clients, and player manager, and has a static interface, the NetworkServerSimple class is a pure network server with no game related functionality. It also has no static interface or singleton, so more than one instance can exist in a process at a time.

The NetworkServer class uses an instance of NetworkServerSimple internally to do connection management.

Properties

Property:	*Function:*
connections	The set of active connections to remote clients. This is a sparse array where NetworkConnect objects reside at the index of their connectionId. There may be nulls in this array for recently closed connections. The connection at index zero may be the connection from the local client.
handlers	The set of registered message handler function.
networkConnectionClass	The type of NetworkConnection object to create for new connections.
hostTopology	The host topology object that this server used to configure the transport layer.
listenPort	The network port that the server is listening on.
serverHostId	The transport layer hostId associated with this server instance.

UnityWebRequest

Leave feedback

UnityWebRequest provides a modular system for composing HTTP requests and handling HTTP responses. The primary goal of the UnityWebRequest system is to allow Unity games to interact with web browser back-ends. It also supports high-demand features such as chunked HTTP requests, streaming POST/PUT operations, and full control over HTTP headers and verbs.

The system consists of two layers:

A High-Level API (HLAPI) wraps the Low-Level API and provides a convenient interface for performing common operations
A Low-Level API (LLAPI) provides maximum flexibility for more advanced users

Supported platforms

The UnityWebRequest system supports most Unity platforms:

All versions of the Editor and Standalone players
WebGL
Mobile platforms: **iOS**, Android
Universal Windows Platform
PS4 and PSVita
XboxOne
Nintendo Switch

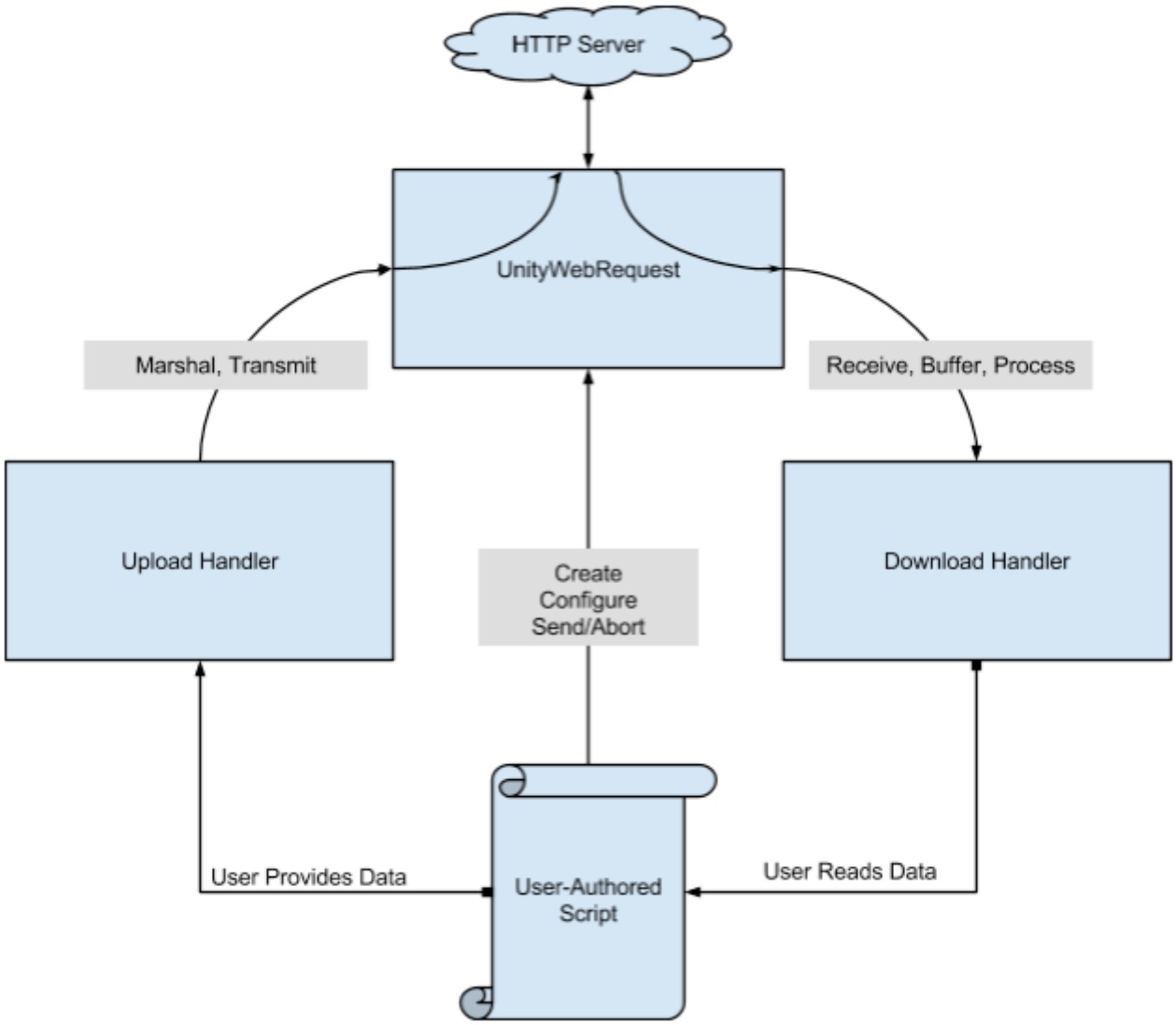
Architecture

The UnityWebRequest ecosystem breaks down an HTTP transaction into three distinct operations:

Supplying data to the server
Receiving data from the server
HTTP flow control (for example, redirects and error handling)

To provide a better interface for advanced users, these operations are each governed by their own objects:

An `UploadHandler` object handles transmission of data to the server
A `DownloadHandler` object handles receipt, buffering and postprocessing of data received from the server
A `UnityWebRequest` object manages the other two objects, and also handles HTTP flow control. This object is where custom headers and URLs are defined, and where error and redirect information is stored.



For any HTTP transaction, the normal code flow is:

Create a Web Request object
Configure the Web Request object
Set custom headers
Set HTTP verb (such as GET, POST, HEAD - custom verbs are permitted on all platforms except for Android)
Set URL
(Optional) Create an Upload Handler and attach it to the Web Request
Provide data to be uploaded
Provide HTTP form to be uploaded
(Optional) Create a Download Handler and attach it to the Web Request
Send the Web Request
If inside a coroutine, you may Yield the result of the `Send()` call to wait for the request to complete
(Optional) Read received data from the Download Handler
(Optional) Read error information, HTTP status code and response headers from the UnityWebRequest object

- 2017–05–16 Page amended with no editorial review

Common operations: using the HLAPI

Leave feedback

This section details the options available in the High-Level API and the scenarios they are intended to address.

Retrieving text or binary data from an HTTP server (GET)
Retrieving a Texture from an HTTP server (GET)
Downloading an AssetBundle from an HTTP server (GET)
Sending a form to an HTTP server (POST)
Uploading raw data to an HTTP server (PUT)

Retrieving text or binary data from an HTTP Server (GET)

Leave feedback

To retrieve simple data such as textual data or binary data from a standard HTTP or HTTPS web server, use the `UnityWebRequest.GET` call. This function takes a single string as an argument, with the string specifying the URL from which data is retrieved.

This function is analogous to the standard WWW constructor:

```
WWW myWww = new WWW("http://www.myserver.com/foo.txt");
// ... is analogous to ...
UnityWebRequest myWr = UnityWebRequest.Get("http://www.myserver.com/foo.txt");
```

Details

This function creates a `UnityWebRequest` and sets the target URL to the string argument. It sets no other custom flags or headers.
By default, this function attaches a standard `DownloadHandlerBuffer` to the `UnityWebRequest`. This handler buffers the data received from the server and make it available to your **scripts** when the request is complete.
By default, this function attaches no `UploadHandler` to the `UnityWebRequest`. You can attach one manually if you wish.

Example

```
using UnityEngine;
using System.Collections;
using UnityEngine.Networking;

public class MyBehaviour : MonoBehaviour {
    void Start() {
        StartCoroutine(GetText());
    }

    IEnumerator GetText() {
        UnityWebRequest www = UnityWebRequest.Get("http://www.my-server.com");
        yield return www.SendWebRequest();

        if(www.isNetworkError || www.isHttpError) {
            Debug.Log(www.error);
        }
        else {
            // Show results as text
            Debug.Log(www.downloadHandler.text);

            // Or retrieve results as binary data
            byte[] results = www.downloadHandler.data;
        }
```

```
        }
    }
```

Retrieving a Texture from an HTTP Server (GET)

Leave feedback

To retrieve a Texture file from a remote server, you can use `UnityWebRequest.Texture`. This function is very similar to `UnityWebRequest.GET` but is optimized for downloading and storing textures efficiently.

This function takes a single string as an argument. The string specifies the URL from which you wish to download an image file for use as a Texture.

Details

This function creates a `UnityWebRequest` and sets the target URL to the string argument. This function sets no other flags or custom headers.
This function attaches a `DownloadHandlerTexture` object to the `UnityWebRequest`. DownloadHandlerTexture is a specialized Download Handler which is optimized for storing images which are to be used as Textures in the Unity Engine. Using this class significantly reduces memory reallocation compared with downloading raw bytes and creating a Texture manually in script.
By default, this function does not attach an Upload Handler. You can add one manually if you wish.

Example

```
using UnityEngine;
using System.Collections;
using UnityEngine.Networking;

public class MyBehaviour : MonoBehaviour {
    void Start() {
        StartCoroutine(GetTexture());
    }

    IEnumerator GetTexture() {
        UnityWebRequest www = UnityWebRequestTexture.GetTexture("http://www.my-s
        yield return www.SendWebRequest();

        if(www.isNetworkError || www.isHttpError) {
            Debug.Log(www.error);
        }
        else {
            Texture myTexture = ((DownloadHandlerTexture)www.downloadHandler).te
        }
    }
}
```

Alternatively, you can implement GetTexture using a helper getter:

```
IEnumerator GetTexture() {
    UnityWebRequest www = UnityWebRequestTexture.GetTexture("http://www.my-s
    yield return www.SendWebRequest();

    Texture myTexture = DownloadHandlerTexture.GetContent(www);
}
```

Downloading an AssetBundle from an HTTP server (GET)

Leave feedback

To download an AssetBundle from a remote server, you can use `UnityWebRequest.GetAssetBundle`. This function streams data into an internal buffer, which decodes and decompresses the AssetBundle's data on a worker thread.

The function's arguments take several forms. In its simplest form, it takes only the URL from which the AssetBundle should be downloaded. You may optionally provide a checksum to verify the integrity of the downloaded data.

Alternately, if you wish to use the AssetBundle caching system, you may provide either a version number or a Hash128 data structure. These are identical to the version numbers or `Hash128 objects` provided to the old system via `WWW.LoadFromCacheOrDownload`.

Details

This function creates a `UnityWebRequest` and sets the target URL to the supplied URL argument. It also sets the HTTP verb to `GET`, but sets no other flags or custom headers.
This function attaches a `DownloadHandlerAssetBundle` to the `UnityWebRequest`. This download handler has a special `assetBundle` property, which can be used to extract the AssetBundle once enough data has been downloaded and decoded to permit access to the resources inside the AssetBundle.
If you supply a version number or `Hash128` object as arguments, it also passes those arguments to the `DownloadHandlerAssetBundle`. The download handler then employs the caching system.

Example

```
using UnityEngine;
using UnityEngine.Networking;
using System.Collections;

public class MyBehaviour : MonoBehaviour {
    void Start() {
        StartCoroutine(GetAssetBundle());
    }

    IEnumerator GetAssetBundle() {
        UnityWebRequest www = UnityWebRequest.GetAssetBundle("http://www.my-serv
        yield return www.SendWebRequest();

        if(www.isNetworkError || www.isHttpError) {
            Debug.Log(www.error);
        }
        else {
            AssetBundle bundle = DownloadHandlerAssetBundle.GetContent(www);
```

```
            }
        }
    }
```

Sending a form to an HTTP server (POST)

Leave feedback

There are two primary functions for sending data to a server formatted as a HTML form. If you are migrating over from the WWW system, see Using WWWForm, below.

Using IMultipartFormSection

To provide greater control over how you specify your form data, the `UnityWebRequest` system contains a user-implementable `IMultipartFormSection` interface. For standard applications, Unity also provides default implementations for data and file sections: `MultipartFormDataSection` and `MultipartFormFileSection`.

An overload of `UnityWebRequest.POST` accepts, as a second parameter, a List argument, whose members must all be `IMultipartFormSections`. The function signature is:

```
WebRequest.Post(string url, List<IMultipartFormSection> formSections);
```

Details

This function creates a `UnityWebRequest` and sets the target URL to the first string parameter. It also sets the Content-Type header of the `UnityWebRequest` appropriately for the form data specified in the list of `IMultipartFormSection` objects.
This function, by default, attaches a `DownloadHandlerBuffer` to the `UnityWebRequest`. This is for convenience - you can use this to check your server's replies.
Similar to the `WWWForm POST` function, this HLAPI function calls each supplied `IMultipartFormSection` in turn and formats them into a standard multipart form as specified in RFC 2616.
The preformatted form data is stored in a standard `UploadHandlerRaw` object, which is then attached to the `UnityWebRequest`. As a result, changes to the `IMultipartFormSection` objects performed after the `UnityWebRequest.POST` call are not reflected in the data sent to the server.

Example

```
using UnityEngine;
using UnityEngine.Networking;
using System.Collections;

public class MyBehavior : MonoBehaviour {
    void Start() {
        StartCoroutine(Upload());
    }

    IEnumerator Upload() {
```

```
        List<IMultipartFormSection> formData = new List<IMultipartFormSection>()
        formData.Add( new MultipartFormDataSection("field1=foo&field2=bar") );
        formData.Add( new MultipartFormFileSection("my file data", "myfile.txt")

        UnityWebRequest www = UnityWebRequest.Post("http://www.my-server.com/myf
        yield return www.SendWebRequest();

        if(www.isNetworkError || www.isHttpError) {
            Debug.Log(www.error);
        }
        else {
            Debug.Log("Form upload complete!");
        }
    }
}
```

Using WWWForm (Legacy function)

To help migrate from the WWW system, the UnityWebRequest system permits you to use the old WWWForm object to provide form data.

In this case, the function signature is:

```
WebRequest.Post(string url, WWWForm formData);
```

Details

This function creates a new `UnityWebRequest` and sets the target URL to the first string argument's value. It also reads any custom headers generated by the `WWWForm` argument (such as Content-Type) and copies them into the `UnityWebRequest`.
This function, by default, attaches a `DownloadHandlerBuffer` to the `UnityWebRequest`. This is for convenience - you can use this to check your server's replies.
This function reads the raw data generated by the `WWWForm object` and buffers it in an `UploadHandlerRaw` object, which is attached to the `UnityWebRequest`. Therefore, changes to the `WWWForm` object after calling `UnityWebRequest.POST` do not alter the contents of the `UnityWebRequest`.

Example

```
using UnityEngine;
using System.Collections;
```

```
public class MyBehavior : public MonoBehaviour {
    void Start() {
        StartCoroutine(Upload());
    }

    IEnumerator Upload() {
        WWWForm form = new WWWForm();
        form.AddField("myField", "myData");

        UnityWebRequest www = UnityWebRequest.Post("http://www.my-server.com/myf
        yield return www.SendWebRequest();

        if(www.isNetworkError || www.isHttpError) {
            Debug.Log(www.error);
        }
        else {
            Debug.Log("Form upload complete!");
        }
    }
}
```

Uploading raw data to an HTTP server (PUT)

Leave feedback

Some modern web applications prefer that files be uploaded via the HTTP PUT verb. For this scenario, Unity provides the `UnityWebRequest.PUT` function.

This function takes two arguments. The first argument is a string and specifies the target URL for the request. The second argument may be either a string or a byte array, and specifies the payload data to be sent to the server.

Function signatures:

```
WebRequest.Put(string url, string data);
WebRequest.Put(string url, byte[] data);
```

Details

This function creates a `UnityWebRequest` and sets the content type to `application/octet-stream`.
This function attaches a standard `DownloadHandlerBuffer` to the `UnityWebRequest`. As with the POST functions, you can use this to return result data from your applications.
This function stores the input upload data in a standard `UploadHandlerRaw` object and attaches it to the `UnityWebRequest`. As a result, if using the `byte[]` function, changes made to the byte array performed after the `UnityWebRequest.PUT` call are not reflected in the data uploaded to the server.

Example

```
using UnityEngine;
using UnityEngine.Networking;
using System.Collections;

public class MyBehavior : MonoBehaviour {
    void Start() {
        StartCoroutine(Upload());
    }

    IEnumerator Upload() {
        byte[] myData = System.Text.Encoding.UTF8.GetBytes("This is some test da
        UnityWebRequest www = UnityWebRequest.Put("http://www.my-server.com/uplo
        yield return www.SendWebRequest();

        if(www.isNetworkError || www.isHttpError) {
            Debug.Log(www.error);
```

```
        }
        else {
            Debug.Log("Upload complete!");
        }
    }
}
```

Advanced operations: Using the LLAPI

Leave feedback

While the HLAPI is designed to minimize boilerplate code, the Low-Level API (LLAPI) is designed to permit maximum flexibility. In general, using the LLAPI involves creating UnityWebRequests then creating appropriate `DownloadHandlers` or `UploadHandlers` and attaching them to your `UnityWebRequests`.

This section details the options available in the Low-Level API and the scenarios they are intended to address:

Creating UnityWebRequests
Creating UploadHandlers
Creating DownloadHandlers

Note that the HLAPI and LLAPI are not mutually exclusive. You can always customize UnityWebRequest objects created via the HLAPI if you need to tweak a common scenario.

For full details on each of the objects described in this section, please refer to the Unity Scripting API.

Creating UnityWebRequests

Leave feedback

WebRequests can be instantiated like any other object. Two constructors are available:

The standard, parameter-less constructor creates a new UnityWebRequest with all settings blank or default. The target URL is not set, no custom headers are set, and the redirect limit is set to 32.
The second constructor takes a string argument. It assigns the UnityWebRequest's target URL to the value of the string argument, and is otherwise identical to the parameter-less constructor.
Multiple other properties are available for setting up, tracking status and checking result or UnityWebRequest.

Example

```
UnityWebRequest wr = new UnityWebRequest(); // Completely blank
UnityWebRequest wr2 = new UnityWebRequest("http://www.mysite.com"); // Target UR

// the following two are required to web requests to work
wr.url = "http://www.mysite.com";
wr.method = UnityWebRequest.kHttpVerbGET;   // can be set to any custom method,

wr.useHttpContinue = false;
wr.chunkedTransfer = false;
wr.redirectLimit = 0;  // disable redirects
wr.timeout = 60;       // don't make this small, web requests do take some time
```

Creating UploadHandlers

Leave feedback

Currently, only one type of upload handler is available: `UploadHandlerRaw`. This class accepts a data buffer at construction time. This buffer is copied internally into native code memory and then used by the `UnityWebRequest` system when the remote server is ready to accept body data.

Upload Handlers also accept a Content Type string. This string is used for the value of the UnityWebRequest's `Content-Type` header if you set no `Content-Type` header on the UnityWebRequest itself. If you manually set a `Content-Type` header on the UnityWebRequest object, the `Content-Type` on the Upload Handler object is ignored.

If you do not set a `Content-Type` on either the UnityWebRequest or the `UploadHandler`, the system defaults to setting a `Content-Type` of `application/octet-stream`.

`UnityWebRequest` has a property `disposeUploadHandlerOnDispose`, which defaults to true. If this property is true, when UnityWebRequest object is disposed, Dispose() will also be called on attached upload handler rendering it useless. If you keep a reference to upload handler longer than the reference to UnityWebRequest, you should set disposeUploadHandlerOnDispose to false.

Example

```
byte[] payload = new byte[1024];
// ... fill payload with data ...

UnityWebRequest wr = new UnityWebRequest("http://www.mysite.com/data-upload");
UploadHandler uploader = new UploadHandlerRaw(payload);

// Sends header: "Content-Type: custom/content-type";
uploader.contentType = "custom/content-type";

wr.uploadHandler = uploader;
```

Creating DownloadHandlers

Leave feedback

There are several types of `DownloadHandlers`:

`DownloadHandlerBuffer` is used for simple data storage.
`DownloadHandlerFile` is used for downloading and saving file to disk with low memory footprint.
`DownloadHandlerTexture` is used for downloading images.
`DownloadHandlerAssetBundle` is used for fetching AssetBundles.
`DownloadHandlerAudioClip` is used for downloading audio files.
`DownloadHandlerMovieTexture` is used for downloading video files. It is recommended that you use VideoPlayer for video download and movie playback since MovieTexture is soon to be deprecated.
`DownloadHandlerScript` is a special class. On its own, it does nothing. However, this class can be inherited by a user-defined class. This class receives callbacks from the UnityWebRequest system, which can then be used to perform completely custom handling of data as it arrives from the network.
The APIs are similar to `DownloadHandlerTexture`'s interface.

`UnityWebRequest` has a property `disposeDownloadHandlerOnDispose`, which defaults to true. If this property is true, when UnityWebRequest object is disposed, Dispose() will also be called on attached download handler rendering it useless. If you keep a reference to download handler longer than the reference to UnityWebRequest, you should set disposeDownloadHandlerOnDispose to false.

DownloadHandlerBuffer

This Download Handler is the simplest, and handles the majority of use cases. It stores received data in a native code buffer. When the download is complete, you can access the buffered data either as an array of bytes or as a text string.

Example

```
using UnityEngine;
using UnityEngine.__Networking__;
using System.Collections;


public class MyBehaviour : MonoBehaviour {
    void Start() {
        StartCoroutine(GetText());
    }

    IEnumerator GetText() {
        UnityWebRequest www = new UnityWebRequest("http://www.my-server.com");
        www.downloadHandler = new DownloadHandlerBuffer();
        yield return www.SendWebRequest();
```

```
        if(www.isNetworkError || www.isHttpError) {
            Debug.Log(www.error);
        }
        else {
            // Show results as text
            Debug.Log(www.downloadHandler.text);

            // Or retrieve results as binary data
            byte[] results = www.downloadHandler.data;
        }
    }
}
```

DownloadHandlerFile

This is a special download handler for large files. It writes downloaded bytes directly to file, so the memory usage is low regardless of the size of the file being downloaded. The distinction from other download handlers is that you cannot get data out of this one, all data is saved to a file.

Example

```
using System.Collections;
using System.IO;
using UnityEngine;
using UnityEngine.Networking;

public class FileDownloader : MonoBehaviour {

    void Start () {
        StartCoroutine(DownloadFile());
    }

    IEnumerator DownloadFile() {
        var uwr = new UnityWebRequest("https://unity3d.com/", UnityWebRequest.kH
        string path = Path.Combine(Application.persistentDataPath, "unity3d.html
        uwr.downloadHandler = new DownloadHandlerFile(path);
        yield return uwr.SendWebRequest();
        if (uwr.isNetworkError || uwr.isHttpError)
            Debug.LogError(uwr.error);
        else
            Debug.Log("File successfully downloaded and saved to " + path);
    }
}
```

DownloadHandlerTexture

Instead of using a `DownloadHandlerBuffer` to download an image file and then creating a texture from the raw bytes using `Texture.LoadImage`, it's more efficient to use `DownloadHandlerTexture`.

This Download Handler stores received data in a `UnityEngine.Texture`. On download completion, it decodes JPEGs and PNGs into valid `UnityEngine.Texture objects`. Only one copy of the `UnityEngine.Texture` is created per `DownloadHandlerTexture` object. This reduces performance hits from garbage collection. The handler performs buffering, decompression and texture creation in native code. Additionally, decompression and texture creation are performed on a worker thread instead of the main thread, which can improve frame time when loading large textures.

Finally, `DownloadHandlerTexture` only allocates managed memory when finally creating the Texture itself, which eliminates the garbage collection overhead associated with performing the byte-to-texture conversion in script.

Example

The following example downloads a PNG file from the internet, converts it to a Sprite, and assigns it to an image:

```
using UnityEngine;
using UnityEngine.__UI__;
using UnityEngine.Networking;
using System.Collections;

[RequireComponent(typeof(UnityEngine.UI.Image))]
public class ImageDownloader : MonoBehaviour {
    UnityEngine.UI.Image _img;

    void Start () {
        _img = GetComponent<UnityEngine.UI.Image>();
        Download("http://www.mysite.com/myimage.png");
    }

    public void Download(string url) {
        StartCoroutine(LoadFromWeb(url));
    }

    IEnumerator LoadFromWeb(string url)
    {
        UnityWebRequest wr = new UnityWebRequest(url);
        DownloadHandlerTexture texDl = new DownloadHandlerTexture(true);
        wr.downloadHandler = texDl;
```

```
        yield return wr.SendWebRequest();
        if(!(wr.isNetworkError || wr.isHttpError)) {
            Texture2D t = texDl.texture;
            __Sprite__ s = __Sprite__.Create(t, new Rect(0, 0, t.width, t.height
                                        Vector2.zero, 1f);
            _img.sprite = s;
        }
    }
}
```

DownloadHandlerAssetBundle

The advantage to this specialized Download Handler is that it is capable of streaming data to Unity's AssetBundle system. Once the AssetBundle system has received enough data, the AssetBundle is available as a `UnityEngine.AssetBundle` object. Only one copy of the `UnityEngine.AssetBundle` object is created. This considerably reduces run-time memory allocation as well as the memory impact of loading your AssetBundle. It also allows AssetBundles to be partially used while not fully downloaded, so you can stream Assets.

All downloading and decompression occurs on worker threads.

AssetBundles are downloaded via a `DownloadHandlerAssetBundle` object, which has a special `assetBundle` property to retrieve the AssetBundle.

Due to the way the AssetBundle system works, all AssetBundle must have an address associated with them. Generally, this is the nominal URL at which they're located (meaning the URL before any redirects). In almost all cases, you should pass in the same URL as you passed to the UnityWebRequest. When using the High Level API (HLAPI), this is done for you.

Example

```
using UnityEngine;
using UnityEngine.Networking;
using System.Collections;

public class MyBehaviour : MonoBehaviour {
    void Start() {
        StartCoroutine(GetAssetBundle());
    }

    IEnumerator GetAssetBundle() {
        UnityWebRequest www = new UnityWebRequest("http://www.my-server.com");
        DownloadHandlerAssetBundle handler = new DownloadHandlerAssetBundle(www.
        www.downloadHandler = handler;
        yield return www.SendWebRequest();
```

```
        if(www.isNetworkError || www.isHttpError) {
            Debug.Log(www.error);
        }
        else {
            // Extracts AssetBundle
            AssetBundle bundle = handler.assetBundle;
        }
    }
}
```

DownloadHandlerAudioClip

This download handler is optimized to for downloading audio files. Instead of downloading raw bytes using `DownloadHandlerBuffer` and then creating `AudioClip` out of them, you can use this download handler to do it in a more convenient way.

Example

```
using System.Collections;
using UnityEngine;
using UnityEngine.Networking;

public class AudioDownloader : MonoBehaviour {

    void Start () {
        StartCoroutine(GetAudioClip());
    }

    IEnumerator GetAudioClip() {
        using (var uwr = UnityWebRequestMultimedia.GetAudioClip("http://myserver
            yield return uwr.SendWebRequest();
            if (uwr.isNetworkError || uwr.isHttpError) {
                Debug.LogError(uwr.error);
                yield break;
            }

            AudioClip clip = DownloadHandlerAudioClip.GetContent(uwr);
            // use audio clip
        }
    }
}
```

DownloadHandlerMovieTexture

Note: MovieTexture will be deprecated. You should use VideoPlayer for video download and movie playback.

This download handler is optimized to for downloading video files. Instead of downloading raw bytes using `DownloadHandlerBuffer` and then creating `MovieTexture` out of them, you can use this download handler to do it in a more convenient way.

Example

```
using System.Collections;
using UnityEngine;
using UnityEngine.Networking;

public class MovieDownloader : MonoBehaviour {

    void Start () {
        StartCoroutine(GetAudioClip());
    }

    IEnumerator GetAudioClip() {
        using (var uwr = UnityWebRequestMultimedia.GetMovieTexture("http://myser
            yield return uwr.SendWebRequest();
            if (uwr.isNetworkError || uwr.isHttpError) {
                Debug.LogError(uwr.error);
                yield break;
            }

            MovieTexture movie = DownloadHandlerMovieTexture.GetContent(uwr);
            // use movie texture
        }
    }
}
```

DownloadHandlerScript

For users who require full control over the processing of downloaded data, Unity provides the `DownloadHandlerScript` class.

By default, instances of this class do nothing. However, if you derive your own classes from `DownloadHandlerScript`, you may override certain functions and use them to receive callbacks as data arrives from the network.

Note: The actual downloads occur on a worker thread, but all `DownloadHandlerScript` callbacks operate on the main thread. Avoid performing computationally heavy operations during these callbacks.

Functions to override

ReceiveContentLength()

```
protected void ReceiveContentLength(long contentLength);
```

This function is called when the Content-Length header is received. Note that this callback may occur multiple times if your server sends one or more redirect responses over the course of processing your UnityWebRequest.

OnContentComplete()

```
protected void OnContentComplete();
```

This function is called when the UnityWebRequest has fully downloaded all data from the server, and has forwarded all received data to the ReceiveData callback.

ReceiveData()

```
protected bool ReceiveData(byte[] data, long dataLength);
```

This function is called after data has arrived from the remote server, and is called once per frame. The `data` argument contains the raw bytes received from the remote server, and `dataLength` indicates the length of new data in the data array.

When not using pre-allocated data buffers, the system creates a new byte array each time it calls this callback, and `dataLength` is always equal to `data.Length`. When using pre-allocated data buffers, the data buffer is reused, and `dataLength` must be used to find the number of updated bytes.

This function requires a return value of either **true** or **false**. If you return **false**, the system immediately aborts the UnityWebRequest. If you return **true**, processing continues normally.

Avoiding garbage collection overhead

Many of Unity's more advanced users are concerned with reducing CPU spikes due to garbage collection. For these users, the UnityWebRequest system permits the pre-allocation of a managed-code byte array, which is used to deliver downloaded data to DownloadHandlerScript's `ReceiveData` callback.

Using this function completely eliminates managed-code memory allocation when using DownloadHandlerScript-derived classes to capture downloaded data.

To make a `DownloadHandlerScript` operate with a pre-allocated managed buffer, supply a byte array to the constructor of `DownloadHandlerScript`.

Note: The size of the byte array limits the amount of data delivered to the ReceiveData callback each frame. If your data arrives slowly, over many frames, you may have provided too small of a byte array.

Example

```
using UnityEngine;
using UnityEngine.Networking;
using System.Collections;

public class LoggingDownloadHandler : DownloadHandlerScript {

    // Standard scripted download handler - allocates memory on each ReceiveData

    public LoggingDownloadHandler(): base() {
    }

    // Pre-allocated scripted download handler
    // reuses the supplied byte array to deliver data.
    // Eliminates memory allocation.

    public LoggingDownloadHandler(byte[] buffer): base(buffer) {
    }

    // Required by DownloadHandler base class. Called when you address the 'byte

    protected override byte[] GetData() { return null; }

    // Called once per frame when data has been received from the network.

    protected override bool ReceiveData(byte[] data, int dataLength) {
        if(data == null || data.Length < 1) {
            Debug.Log("LoggingDownloadHandler :: ReceiveData - received a null/e
            return false;
        }

        Debug.Log(string.Format("LoggingDownloadHandler :: ReceiveData - receive
        return true;
    }

    // Called when all data has been received from the server and delivered via
```

```
    protected override void CompleteContent() {
        Debug.Log("LoggingDownloadHandler :: CompleteContent - DOWNLOAD COMPLETE
    }

    // Called when a Content-Length header is received from the server.

    protected override void ReceiveContentLength(int contentLength) {
        Debug.Log(string.Format("LoggingDownloadHandler :: ReceiveContentLength
    }
}
```

Audio

Leave feedback

Unity's Audio features include full 3D spatial sound, real-time mixing and mastering, hierarchies of mixers, snapshots, predefined effects and much more.

Read this section to learn about audio in Unity, including clips, sources, listeners, importing and sound settings.

Related tutorials: Audio

See the Knowledge Base Audio section for tips, tricks and troubleshooting.

Audio Overview

Leave feedback

A game would be incomplete without some kind of audio, be it background music or sound effects. Unity's audio system is flexible and powerful. It can import most standard audio file formats and has sophisticated features for playing sounds in 3D space, optionally with effects like echo and filtering applied. Unity can also record audio from any available microphone on a user's machine for use during gameplay or for storage and transmission.

Basic Theory

In real life, sounds are emitted by objects and heard by listeners. The way a sound is perceived depends on a number of factors. A listener can tell roughly which direction a sound is coming from and may also get some sense of its distance from its loudness and quality. A fast-moving sound source (like a falling bomb or a passing police car) will change in pitch as it moves as a result of the Doppler Effect. Also, the surroundings will affect the way sound is reflected, so a voice inside a cave will have an echo but the same voice in the open air will not.

Audio Sources and Listener

To simulate the effects of position, Unity requires sounds to originate from **Audio Sources** attached to objects. The sounds emitted are then picked up by an **Audio Listener** attached to another object, most often the main **camera**. Unity can then simulate the effects of a source's distance and position from the listener object and play them to the user accordingly. The relative speed of the source and listener objects can also be used to simulate the Doppler Effect for added realism.

Unity can't calculate echoes purely from **scene** geometry but you can simulate them by adding **Audio Filters** to objects. For example, you could apply the Echo filter to a sound that is supposed to be coming from inside a cave. In situations where objects can move in and out of a place with a strong echo, you can add a **Reverb Zone** to the **scene**. For example, your game might involve cars driving through a tunnel. If you place a reverb zone inside the tunnel then the cars' engine sounds will start to echo as they enter and the echo will die down as they emerge from the other side.

The Unity **Audio Mixer** allows you to mix various audio sources, apply effects to them, and perform mastering.

The manual pages for Audio Source, Audio Listener, Audio Mixer, the audio effects and Reverb Zones give more information about the many options and parameters available for getting effects just right.

Working with Audio Assets

Unity can import audio files in **AIFF**, **WAV**, **MP3** and **Ogg** formats in the same way as other **assets**, simply by dragging the files into the Project panel. Importing an audio file creates an **Audio Clip** which can then be dragged to an Audio Source or used from a script. The **Audio Clip** reference page has more details about the import options available for audio files.

For music, Unity also supports tracker modules, which use short audio samples as "instruments" that are then arranged to play tunes. Tracker modules can be imported from **.xm**, **.mod**, **.it**, and **.s3m** files but are otherwise used in much the same way as ordinary audio clips.

Audio Recording

Unity can access the computer's microphones from a script and create Audio Clips by direct recording. The **Microphone** class provides a straightforward API to find available microphones, query their capabilities and start and end a recording session. The script reference page for Microphone has further information and code samples for audio recording.

Audio files

Leave feedback

As with Meshes or Textures, the workflow for **Audio File assets** is designed to be smooth and trouble free. Unity can import almost every common file format but there are a few details that are useful to be aware of when working with Audio Files.

Since Unity 5.0 audio data is separated from the actual AudioClips. The AudioClips merely refer to the files containing the audio data and there are various combinations of options in the AudioClip importer that determine how the clips are loaded at runtime. This means that you have great flexibility for deciding which audio assets should be kept in memory at all times (because you may not be able to predict how often or how fast they will be playing, i.e. footsteps, weapons and impacts), while other assets may be loaded on demand or gradually as the player progresses through the level (speech, background music, ambience loops etc).

When audio is encoded in Unity the main options for how it is stored on disk is either *PCM*, *ADPCM* or *Compressed*. Additionally there are a few platform-specific formats, but they work in similar ways. Unity supports most common formats for importing audio (see the list below) and will import an audio file when it is added to the project. The default mode is *Compressed*, where the audio data is compressed with either Vorbis/MP3 for standalone and mobile platforms, or HEVAG/XMA for PS Vita and **Xbox One**.

See the AudioClip documentation for an extensive description of the **compression** formats and other options available for encoding audio data.

Any Audio File imported into Unity is available from **scripts** as an **Audio Clip** instance, which provides a way for the game runtime of the audio system to access the encoded audio data. The game may access meta-information about the audio data via the AudioClip even before the actual audio data has been loaded. This is possible because the import process has extracted various bits of information such as length, channel count and sample rate from the encoded audio data and stored it in the AudioClip. This can for instance be useful when creating automatic dialogue or music sequencing systems, because the music engine can use the information about the length to schedule music playback before actually loading the data. It also helps reducing memory usage by only keeping the audio clips in memory that are needed at a time.

Supported formats

Format	*Extensions*
MPEG layer 3	.mp3
Ogg Vorbis	.ogg
Microsoft Wave	.wav
Audio Interchange File Format	.aiff / .aif
Ultimate Soundtracker module	.mod
Impulse Tracker module	.it
Scream Tracker module	.s3m
FastTracker 2 module	.xm

See the Audio Overview for more information on using sound in Unity.

Tracker Modules

Leave feedback

Tracker Modules are essentially just packages of audio samples that have been modeled, arranged and sequenced programatically. The concept was introduced in the 1980's (mainly in conjunction with the Amiga computer) and has been popular since the early days of game development and demo culture.

Tracker Module files are similar to MIDI files in many ways. The tracks are scores that contain information about when to play the instruments, and at what pitch and volume and from this, the melody and rhythm of the original tune can be recreated. However, MIDI has a disadvantage in that the sounds are dependent on the sound bank available in the audio hardware, so MIDI music can sound different on different computers. In contrast, tracker modules include high quality PCM samples that ensure a similar experience regardless of the audio hardware in use.

Supported formats

Unity supports the four most common module file formats, namely Impulse Tracker (**.it**), Scream Tracker (**.s3m**), Extended Module File Format (**.xm**), and the original Module File Format (**.mod**).

Benefits of Using Tracker Modules

Tracker module files differ from mainstream PCM formats (**.aif**, **.wav**, **.mp3**, and **.ogg**) in that they can be very small without a corresponding loss of sound quality. A single sound sample can be modified in pitch and volume (and can have other effects applied), so it essentially acts as an "instrument" which can play a tune without the overhead of recording the whole tune as a sample. As a result, tracker modules lend themselves to games, where music is required but where a large file download would be a problem.

Third Party Tools and Further References

Currently, the most popular tools to create and edit Tracker Modules are MilkyTracker for OSX and OpenMPT for Windows. For more information and discussion, please see the blog post .mod in Unity from June 2010.

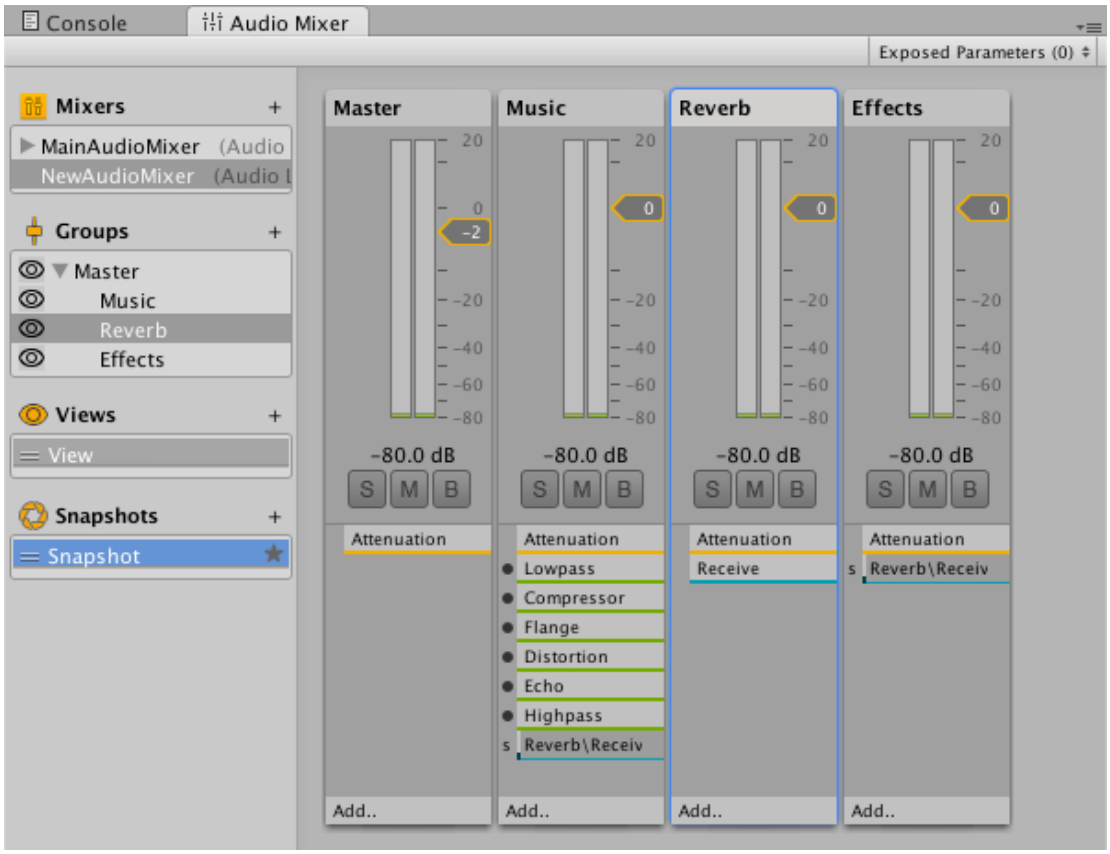
Audio Mixer

Leave feedback

The Unity Audio Mixer allows you to mix various audio sources, apply effects to them, and perform mastering.

Audio Mixer Window

The window displays the Audio Mixer which is basically a tree of Audio Mixer Groups. An Audio Mixer group is essentially a mix of audio, a signal chain which allows you to apply volume attenuation and pitch correction; it allows you to insert effects that process the audio signal and change the parameters of the effects. There is also a send and return mechanism to pass the results from one bus to another.



The Audio Mixer

An Audio Mixer is an **asset**. You can create one or more Audio Mixer and have more than one active at any time. An Audio Mixer always contains a master group. Other groups can then be added to define the structure of the mixer.

How it works

You route the output of an Audio Source to a group within an Audio Mixer. The effects will then be applied to that signal.

The output of an Audio Mixer can be routed into any other group in any other Audio Mixer in a **scene** enabling you to chain up a number of Audio Mixers in a **scene** to produce complex routing, effect processing and snapshot applying.

Snapshots

You can capture the settings of all the parameters in a group as a snapshot. If you create a list of snapshots you can then transition between them in gameplay to create different moods or themes.

Ducking

Ducking allows you to alter the effect of one group based on what is happening in another group. An example might be to reduce the background ambient noise while something else is happening.

Views

Different views can be set up. You can disable the visibility of certain groups within a mixer and set this as a view. You can then transition between views as required.

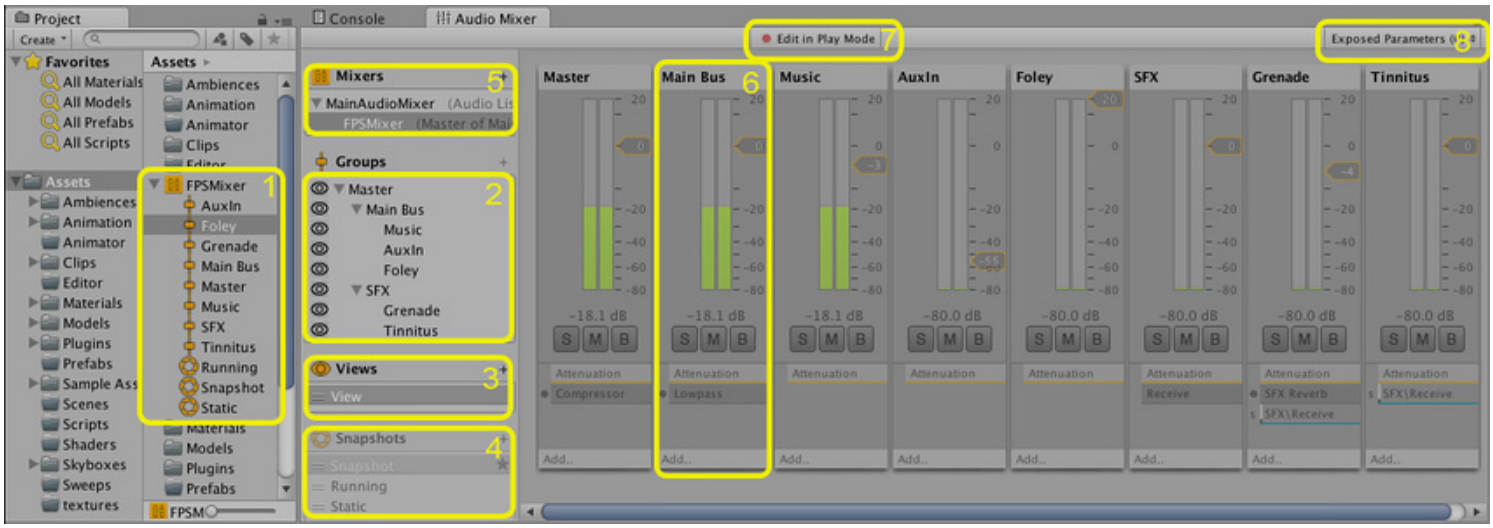
An overview of the concepts and AudioMixer

Leave feedback

The AudioMixer is an **Asset** that can be referenced by AudioSources to provide more complex routing and mixing of the audio signal generated from AudioSources. It does this category based mixing via the AudioGroup hierarchy that is constructed by the user inside the **Asset**.

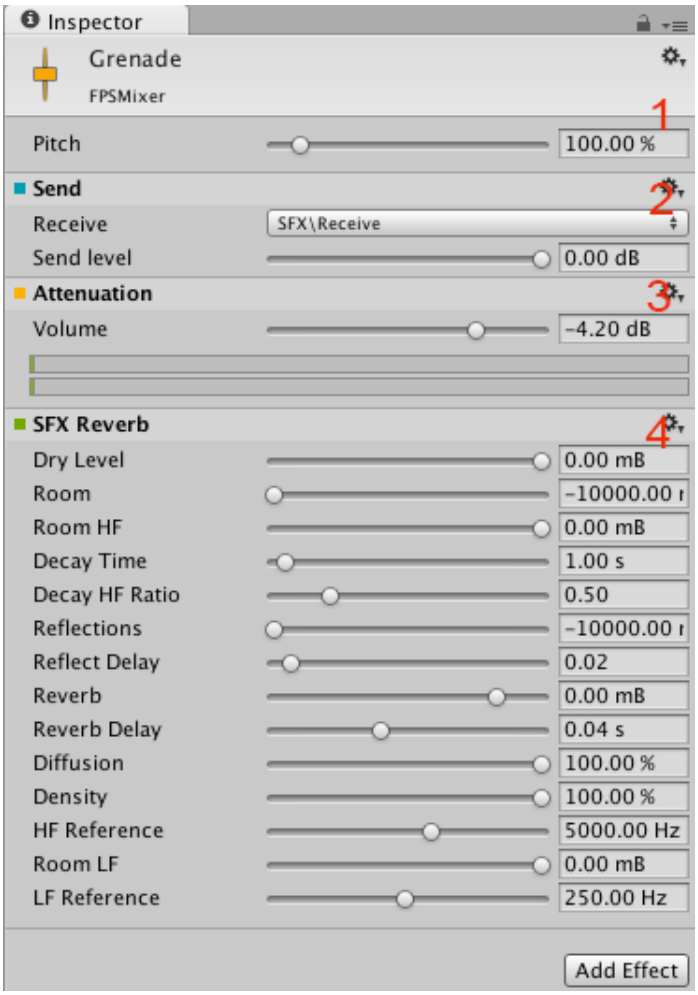
DSP effects and other audio mastering concepts can be applied to the audio signal as it is routed from the AudioSource to the AudioListener.

AudioMixer View



The Asset - Containing all AudioGroups and AudioSnapshots as sub-assets.
The Hierarchy view - This contains the entire mixing hierarchy of AudioGroups within the AudioMixer.
The Mixer Views - This is a list of cached visibility settings of the mixer. Each view only shows a sub-set of the entire hierarchy in the main mixer window.
Snapshots - This is a list of all the AudioSnapshots within the AudioMixer Asset. Snapshots capture the state of all the parameter settings within an AudioMixer, and can be transitioned between at runtime.
The Output AudioMixer - AudioMixers can be routed into AudioGroups of other AudioMixers. This property field allows one to define the output AudioGroup to route this AudioMixer signal into.
AudioGroup Strip View - This shows an overview of an AudioGroup, including the current VU levels, attenuation (volume) settings, Mute, Solo and Effect Bypass settings and the list of DSP effects within the AudioGroup.
Edit In Play Mode - This is a toggle that allows you to edit the AudioMixer during play mode, or prevent edits and allow the game runtime to control the state of the AudioMixer.
Exposed Parameters - This shows a list of Exposed Parameters (any parameter within an AudioMixer can be exposed to script via a string name) and corresponding string names.

AudioMixer Inspector



The Pitch and Ducking settings are present at the top of all AudioGroups.
An example Send Effect, positioned before where the attenuation is applied.
The Attenuation (volume setting) is done here for an AudioGroup. The Attenuation can be applied anywhere in the effect stack. The VU meter here shows the volume levels at that point in the effect stack (different from the VU meters shown in the AudioMixer view which show the levels of the signal as it leaves the AudioGroup.
An example effect with parameters, in this case a Reverb. Parameters can be exposed by right clicking on them and choosing to expose them.

Concepts

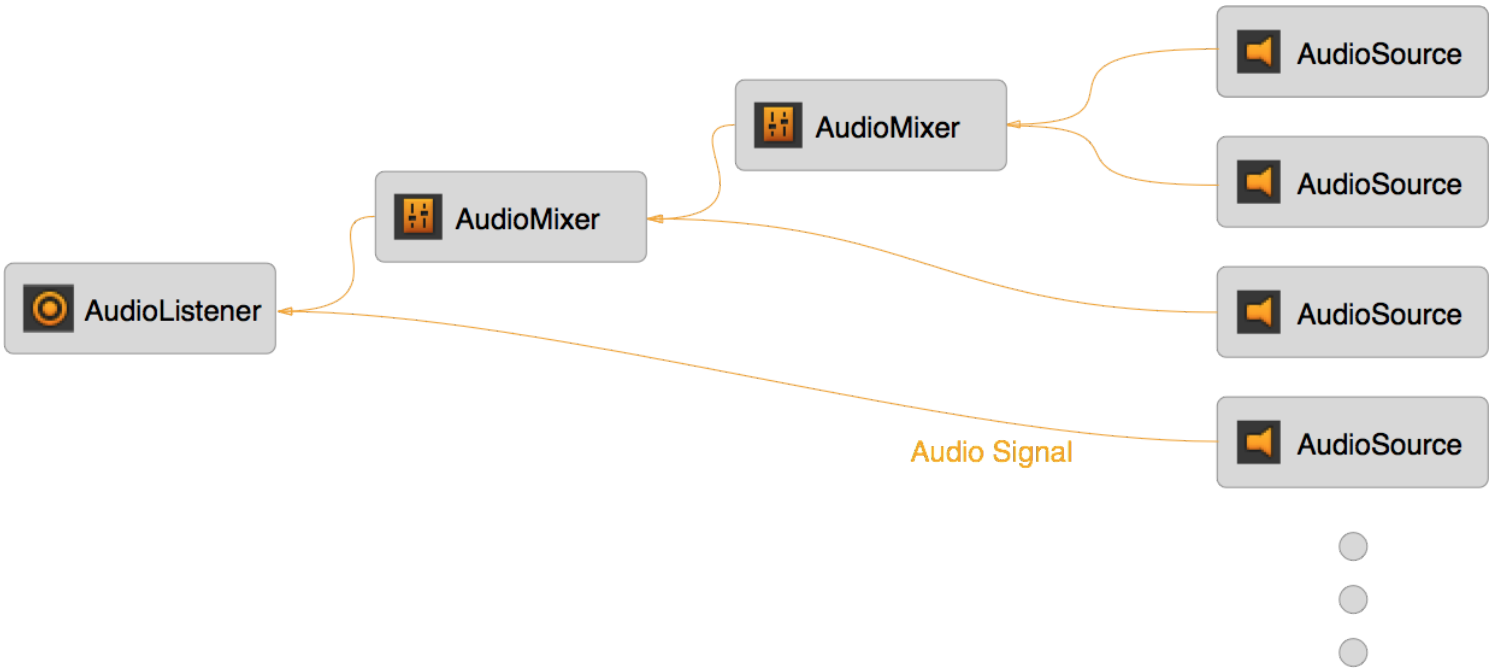
Routing and Mixing

http://en.wikipedia.org/wiki/Audio_mixing

Audio routing is the process of taking a number of input audio signals and outputting 1 or more output signals. The term signal here refers to a continuous stream of digital audio data, which can be broken down into digital audio channels (such as stereo or 5.1 (6 channels)).

Internally there is usually some work on these signals being done, such as mixing, applying effects, attenuation **etc**. For various reasons that will be covered, this is an important aspect of audio processing and this is what the AudioMixer is designed to allow you to do.

With the exception of Sends and Returns (which will be covered later), the AudioMixer contains AudioGroups that allow any number of input signals, mix those signals and have exactly 1 output.



In the land of audio processing, this routing and mixing is usually done ORTHOGONAL to the **scene** graph hierarchy, as audio behaves and designers interact with audio very differently to objects and concepts shown in the **scene**.

In previous versions of Unity, the concept of routing and mixing was not available. Users were able to place AudioSources within the scene, and the audio signal that they produced (via AudioClips for example) was routed directly to the AudioListener, where all the audio signals were mixed together at one point. Remember here that this is happening orthogonal to the scene graph and no matter where your AudioSources are in the scene,

AudioMixers now sit between the AudioSource and the AudioListener in the audio signal processing space and allow you to take the output signal from the AudioSource perform whatever routing and mixing operations they wish until finally all audio is output to the AudioListener and is heard from the speakers.

Why do any of this stuff?

Mixing and routing allows you to categorise the audio within their game into whatever concepts they desire. Once sound is mixed together into these categories, effects and other operations can be applied to these categories as a whole. This is powerful not only applying game logic changes to the various sound categories, but also for allowing designers to tweak the various aspects of the mix to perform whats knows as “Mastering” of the entire soundscape dynamically at runtime.

Relation to 3D spatial attenuation

Some sound concepts are related to the scene graph and the 3D world. The most obvious of those is the application of attenuation based on 3D distance, relative speed to the AudioListener and environmental Reverb effects.

As these operations are related to the scene and not to the categories of sounds in an AudioMixer, the effects are applied at the AudioSource, before the signal enters an AudioMixer. For example, the attenuation applied to an AudioSource based on its distance from the AudioListener is applied to the signal before it leaves the AudioSource and is routed into an AudioMixer.

Sound Categories

As stated above, AudioMixers allow you to effectively categorise types of sounds and do stuff to these categories. This is an important concept, because without such categorisations, the entire soundscape quickly becomes a mess of indistinguishable noise as every sound is played back equally and without any mixing applied to them. With concepts such as ducking, categories of sounds can also influence each other, adding additional richness to the mix.

Examples of operations that designers might want to do on a category are;

Apply attenuation to a group of ambiences.
Trigger a lowpass filter on all the foley sounds in a game, simulating being underwater.
Attenuate all sounds in the game except the Menu music and interaction sounds.
Reduce the volume of all the gun and explosion sounds in a game to ensure that an NPC talking to you can be heard.
etc...

These categories are really quite game specific and vary between different projects, but an example of such categorisation might be described as follows;

All sounds are routed into the “Master” AudioGroup
Into the Master group, there is a category for Music, Menu sounds and all game sounds
The game sounds group is broken down into dialog from NPCs, environmental sounds from ambiences and other foley sounds like gunshots and footsteps
These categories are broken further down as required

The category hierarchy of this layout would look something like this:

Note that the scene graph layout would look nothing like the layout for sound categories.

Moods and Themes of the Mix

Mixing and routing of the games sounds can also be used to create the immersion the designer is looking for. For example, reverb can be applied to all of the game sounds and the music attenuated to create the feeling of being in a cave.

The AudioMixer can be used effectively to create moods within the game. Using concepts such as snapshots (explained later) and other different mixers within a game, the game can transition its mood easily and emote the player into feeling what the designer wishes, which is super powerful in the immersion of the game.

The Global Mix

The AudioMixer is used to control the overall mix of all the sounds within a game. These AudioMixers control the global mix and can be seen as the static singleton mix that sound instances are routed through.

In other worlds, the AudioMixers are always present through the lifetime of a scene, and sound instances are created and destroyed as the game progresses and play through these global AudioMixers.

Snapshots

Snapshots allow you to capture the state of an AudioMixer, and transition between these different states as the game progresses. This is a great way to define moods or themes of the mix and have those moods change as the player progresses through the game.

Snapshots capture the values of all of the parameters within the AudioMixer;

Volume
Pitch
Send Level
Wet Mix Level
Effect Parameters
Combining Snapshots with game logic is a great way to change many aspects of the soundscape.

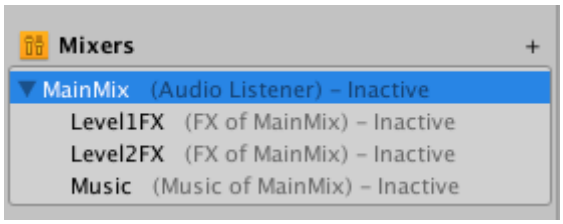
Specifics on the AudioMixer window

Leave feedback

Mixers Panel

The Mixers Panel shows a complete list of all AudioMixers within the project. AudioMixers can be quickly switched between by selecting them within this panel. Routing one AudioMixer into the AudioGroup of another AudioMixer is also performed within this panel.

You can also create new AudioMixers in the project my clicking the '+' icon in the top right of the panel.



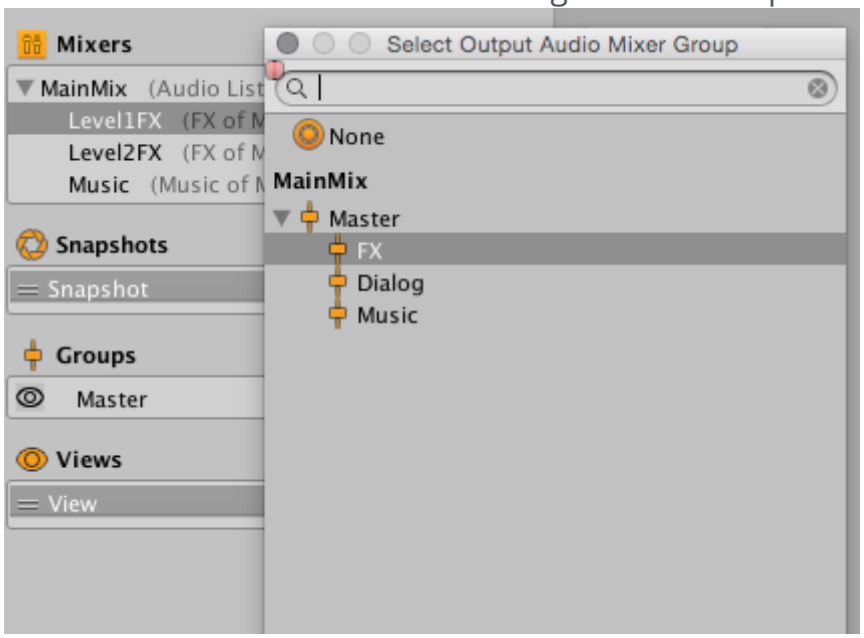
Routing AudioMixers into other AudioMixers

Unity supports having multiple AudioMixers used within a **scene** at once. By default, each AudioMixer outputs the audio signal directly to the AudioListener.

Developers can also choose to route the audio output of an AudioMixer into an AudioGroup of another AudioMixer. This allows for flexible and dynamic routing hierarchies at game runtime.

Routing an AudioMixer into another AudioGroup can be achieved two ways, firstly in the editor within the Mixers Panel, and the other dynamically at runtime using the AudioMixer API.

To change the output of an AudioMixer within the editor, simply click on an AudioMixer within the Mixers Panel and drag it over the top of another AudioMixer.
You will be presented with a dialog allowing you to select the AudioGroup of the target AudioMixer you would like to route into.
Once you select an output AudioGroup, the panel will show the parenting relationship of the AudioMixers. It will also show the target AudioGroup next to the AudioMixers name.



Hierarchy Panel

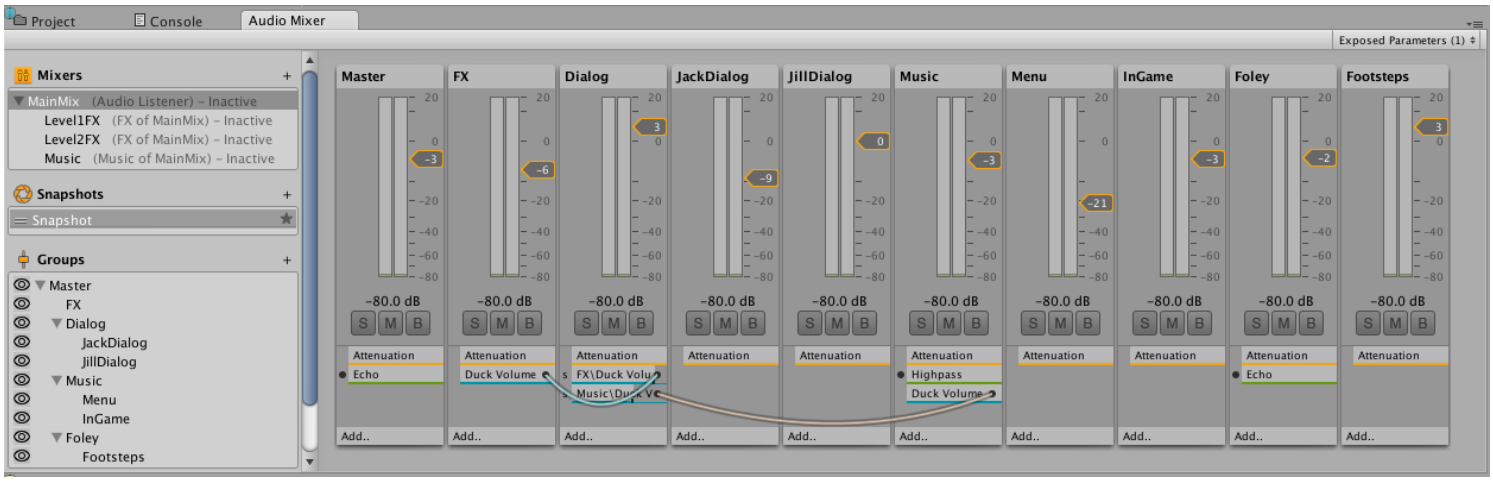
The hierarchy view is where you define the sound categories of an AudioMixer and the mixing structure. As described above, it allows you to define your own custom categories that AudioSources can connect to and play through.

Adding and Configuring AudioGroups within the Hierarchy

Adding and modifying the topology of an AudioMixer is done within the AudioGroup Hierarchy Panel.

Adding a new AudioGroup to the hierarchy can be done in 2 ways;
Right clicking on an existing AudioGroup (there must be at least one in an AudioMixer) and selecting 'Add child group' or 'Add sibling group'.
Selecting an AudioGroup that you would like to add a child to and clicking the '+' icon in the top right of the panel. This will add a new group to the AudioMixer under the selected one.
Changing the topology of an AudioMixer can be done by clicking an AudioGroup within the panel and dragging it over the top of another AudioGroup, this will parent the target AudioGroup above the one selected.
Deleting an AudioGroup (including its children) is achieved 2 ways;
By selecting the group you would like to delete and pressing the Delete key.
By right clicking the group you would like to delete and selecting the "Remove Group (and children)" option.
To duplicate an AudioGroup (and make it a sibling), right click the AudioGroup you wish to duplicate and select "Duplicate Group (and children)". This will duplicate the group and child groups exactly, including effects contained within the groups.
To rename an AudioGroup, right click on the group and select "Rename".

AudioGroup View



The AudioGroup View shows a flat arrangement of the AudioGroups in the AudioMixer. This arrangement is organised horizontally within the view. The groups shown within the AudioGroup View are dictated by the current View selection (covered later).

Each AudioGroup within the view is represented as a vertical “strip”. The layout and look and feel of the strip is common to Digital Audio Workstations and other audio editing packages. This layout is chosen to facilitate the transition of Audio Engineers from a music and video background, as well as serve as a parallel for audio hardware integration.

The strip is made up of title bar, followed by a vertical VU meter which represents the current audio levels through that AudioGroup. Beside the VU meter is a volume selector which allows you to input the AudioGroups attenuation along the same scale as the VU meter, which is represented in dB levels.

Below the VU meter are 3 buttons with the following functionality:

Solo - This toggle will enable you to switch between hearing the entire mix or only the AudioSources that are playing into children of the AudioGroup being Soloed.
Mute - This toggle allows you to switch between including the current AudioGroup in the audible mix or excluding it from being heard in the global mix.
Bypass - This toggle allows you to bypass or enable all the effects present within the AudioGroup.
The AudioGroup also contains a list of DSP Effect Units and the Attenuation Units within the AudioGroup. The attenuation can be applied anywhere within the Effect Unit chain within an AudioGroup and allows you to decide exactly where you would like the volume adjustment to be applied. This is useful for non-linear effects and Send and Receive Units (covered later).

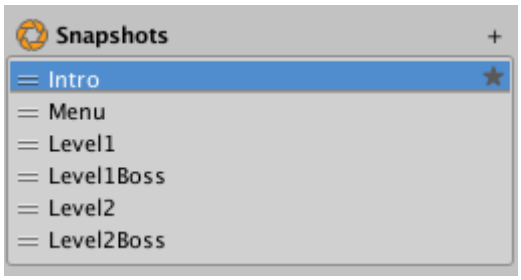
Each Effect Unit slot has the following functionality:

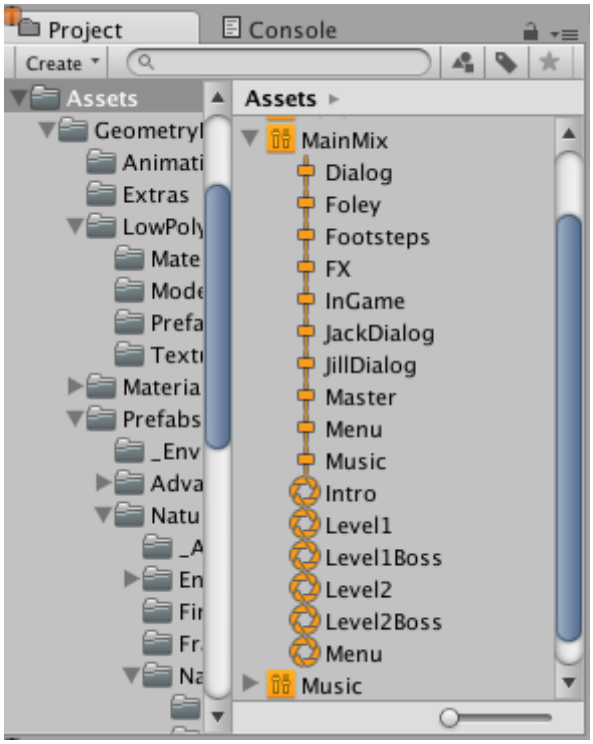
It shows the name of the Effect Unit that is being applied
It shows a circle on the left hand side of the effect that can be toggled to enable or bypass that individual effect.
If you right click the effect and select “Allow Wet Mixing”, the coloured bar on the bottom of the effect slow becomes active, and denotes the amount of wet signal that is being passed through the effect.
Effects can be dragged up and down the AudioGroup to refine their order and also across AudioGroups to move the effect to another AudioGroup.
You can also add new Effect Units by right clicking on an existing effect to add before or after it, or by clicking the “Add..” button at the bottom of the strip.

Snapshot Panel

The Snapshot Panel allows you to create, switch between and tweak different Snapshots within the AudioMixer. There is always at least one snapshot active, and selection of a snapshot within the Snapshot Panel indicates that further edits of the AudioMixer are edits to that snapshot.

Snapshots defined in the Snapshot Panel also show up as sub-assets of the AudioMixer. This allows you to access the snapshots elsewhere in the editor and within **scripts**.





You also define a 'Start Snapshot' (indicated by the star icon on the right hand side of the snapshot list). The Start Snapshot is the snapshot that the AudioMixer will be initialised to when loaded (for example, when the scene starts)

To create a new Snapshot, click the small '+' icon at the top right of the panel. Enter in a name for the new snapshot.
To define a different Start Snapshot, right click on the desired Snapshot and choose "Set as Start Snapshot".

Views Panel

Views allow you to create groups of visible AudioGroups in the AudioMixer. With views, you can create perspectives of interest into the AudioMixer, instead of always being presented with the full hierarchy at all times. Views are purely for workflow optimisation purposes are do not affect runtime setup or performance.

Like the Snapshot Panel, there is always one view selected and currently shown in the AudioGroup View. By default, all AudioGroups are visible in the default view. What is contained within a view is controlled by the Eye Icons in the Hierarchy Panel (see above).

To add a new view to the list of views, click the small '+' icon at the top right of the Views Panel. Enter a name for the new view.
Change the current view by selecting between the list of views in the View Panel.
To remove a view, right click on the view and select 'Delete'
To Duplicate a view (with all of the current view settings, right click on the view and select 'Duplicate'

The ‘Eye’ Icon of an AudioGroup

Each AudioGroup within the hierarchy panel has a small eye icon to the left of the group. This icon serves 2 purposes;

Clicking on the eye icon toggles this AudioGroup’s visibility in the currently selected View.
Right clicking on the eye icon allows you to select from a range of colours to tag this AudioGroup with. Selecting a colour other than “No Colour” will give a small colour tag to the left of the eye icon as well as a colour tag under the AudioGroup’s name in the AudioGroup view. These colour indicators are a great way to visually group different concepts and collections of AudioGroups within the AudioMixer.

AudioGroup Inspector

Leave feedback

Selecting an AudioGroup in the AudioGroup Hierarchy, the AudioGroup View or the Project window (as a sub-asset) will show the **inspector** for that AudioGroup.

The inspector for the AudioGroup consists of a number of elements:

Inspector Header

At the top of the AudioGroup Inspector there is the name of the AudioGroup, along with with the gear dropdown menu common to all Object Inspectors.

The gear menu contains the following functionality:

'Copy all effect settings to all snapshots' - This lets you to copy all of the effect parameter, volume and pitch settings of this AudioGroup to all of the other snapshots present within the AudioMixer. This lets you to quickly make all snapshots 'like this one' for this AudioGroup.
'Toggle CPU usage display' - This toggles CPU performance information for all of the effects present in the AudioGroup inspector. This is used to get an idea of which effects within your DSP setup are consuming the most resources.

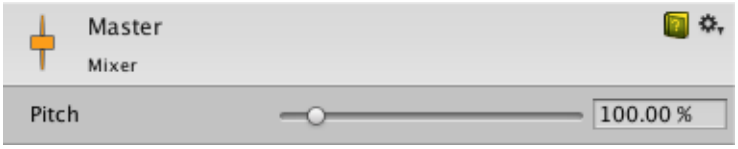
Edit in Playmode

When in Playmode within Unity, the Inspector for an AudioGroup includes a button at the top called "Edit in Playmode". By default, the parameter values of an AudioMixer are not editable in Playmode and are fully controlled by the current snapshot within the game.

Edit in Playmode allows you to override the snapshot system and start making edits to the current snapshot directly during playmode. This is a great way to mix and master a game while playing it in realtime.

Pitch Slider

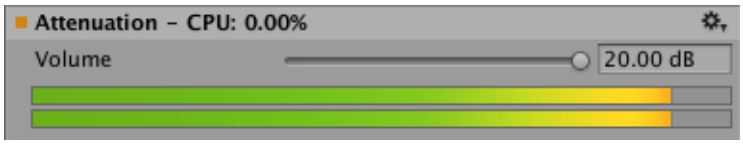
At the top of all AudioGroup Inspectors, there is a slider that defines the pitch of playback through that AudioGroup. To change the pitch, either use the slider or enter the pitch manually into the text field to the right.



Attenuation Unit

Every AudioGroup within an AudioMixer has exactly 1 Attenuation Unit.

The Attenuation Unit is where you can apply attenuation / gain to the audio signal passing through the AudioGroup. The attenuation is computed and applied to the signal 'at the unit' (not combined with other attenuation settings and applied at the voice source). This allows very complex and interesting setups to be created when combined with Sends / Receives and non linear DSP effects. Attenuation can be applied to –80dB (silence) and gain can be applied to +20dB.



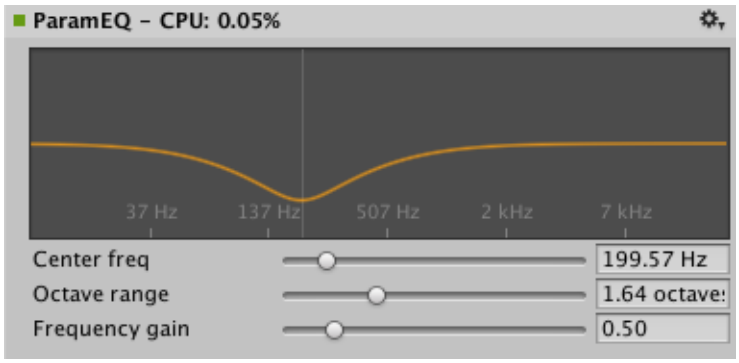
Every Attenuation Unit has a VU meter in the inspector. This meter shows the audio signal levels at that point in the signal chain (just after attenuation is applied). This means that if you have DSP effects or Receives after the Attenuation Unit, the metering information seen in the AudioGroup strip for that AudioGroup will be different to the metering information at the Attenuation Unit. This is a great way to debug the signal chain of an AudioGroup by dragging the Attenuation Unit up and down the processing chain to see the metering at different points.

The VU meter shows both RMS and peak hold values.

To move the Attenuation Unit (or any effect) up or down the signal chain, click on the Unit's header and drag up or down the inspector to reposition it.
To change the attenuation setting, move the slider above the metering or enter in a value in the text box.

Effect Units

Effect Units are general DSP effects that modify the audio signal being played through the AudioGroup, for example Highpass or Reverb. Effect Units can also process side-chain signal information that is sent to it from a Send Unit. The interface for each Effect Unit is different, but for the most part expose a collection of parameters that you can modify to change how the effect is applied to the signal. For example, a Parameter EQ effect has 3 parameters that modify how the signal is processed:



Unity comes with a collection of in-built effects that you can use within an AudioGroup. There is also the ability to create custom DSP effect **plugins** that can be used within an AudioMixer.

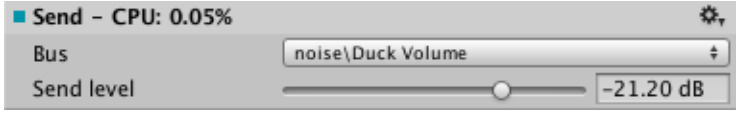
To add an effect to the AudioGroup, click the 'Add Effect' button at the bottom of the AudioGroup Inspector.

To change the ordering of the effect within the AudioGroup, left click the effect header and drag up or down to place it in a different position.

To remove the effect from the AudioGroup, right click on the effect header and select 'Remove this effect'.

Send Units

Sends allow you to diverge the audio signal flow and send a potentially attenuated copy of the signal to be used as a side-chain within another Effect Unit, for example, a side-chain compressor. You can insert Sends anywhere in the signal chain, allowing divergence of signal at any point.

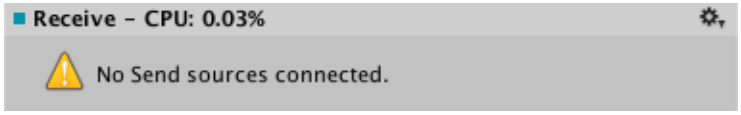


Initially, when Sends are added to an AudioGroup, they do not send to anything, and the Send Level is set to 80dB. To send to another Effect Unit, you must already have an Effect Unit that can accept side-chain signals in the AudioMixer somewhere. Once the destination Effect Unit has been selected, the user needs to increase the Send Level to send signal to the destination.

To add a Send to an AudioGroup, click the 'Add Effect' button at the bottom of the AudioGroup Inspector and choose 'Send'.
To connect a Send to another Effect Unit (capable of receiving signal), choose the destination from the drop down menu in the Send Unit Inspector.
Set the level of signal sent to the destination with "Send Level"

Receive Units

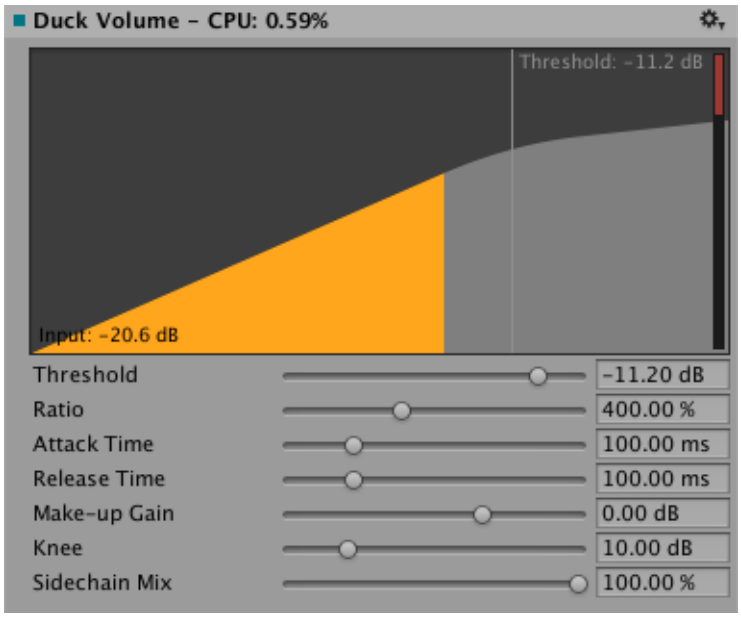
Receives are the signal sinks of Sends, they simply take the audio signal that is sent to them from Sends and mix it with the current signal passing through their AudioGroup. There are no parameters to a Receive.



Note that if you Solo a Receive unit, this will make the sound stop playing. This is by design.

Duck Volume Units

Duck Volume Units allow you to create side-chain **compression** from signal sent from Sends. Duck Volume is a great way to control the attenuation of a signal based on audio being played somewhere else in the AudioMixer.



Duck Volume Units can be added like any other Effect Unit and must have signal sent to them from at least one Send to be useful.

Common Options

Each unit within the AudioGroup Inspector has a number of common features.

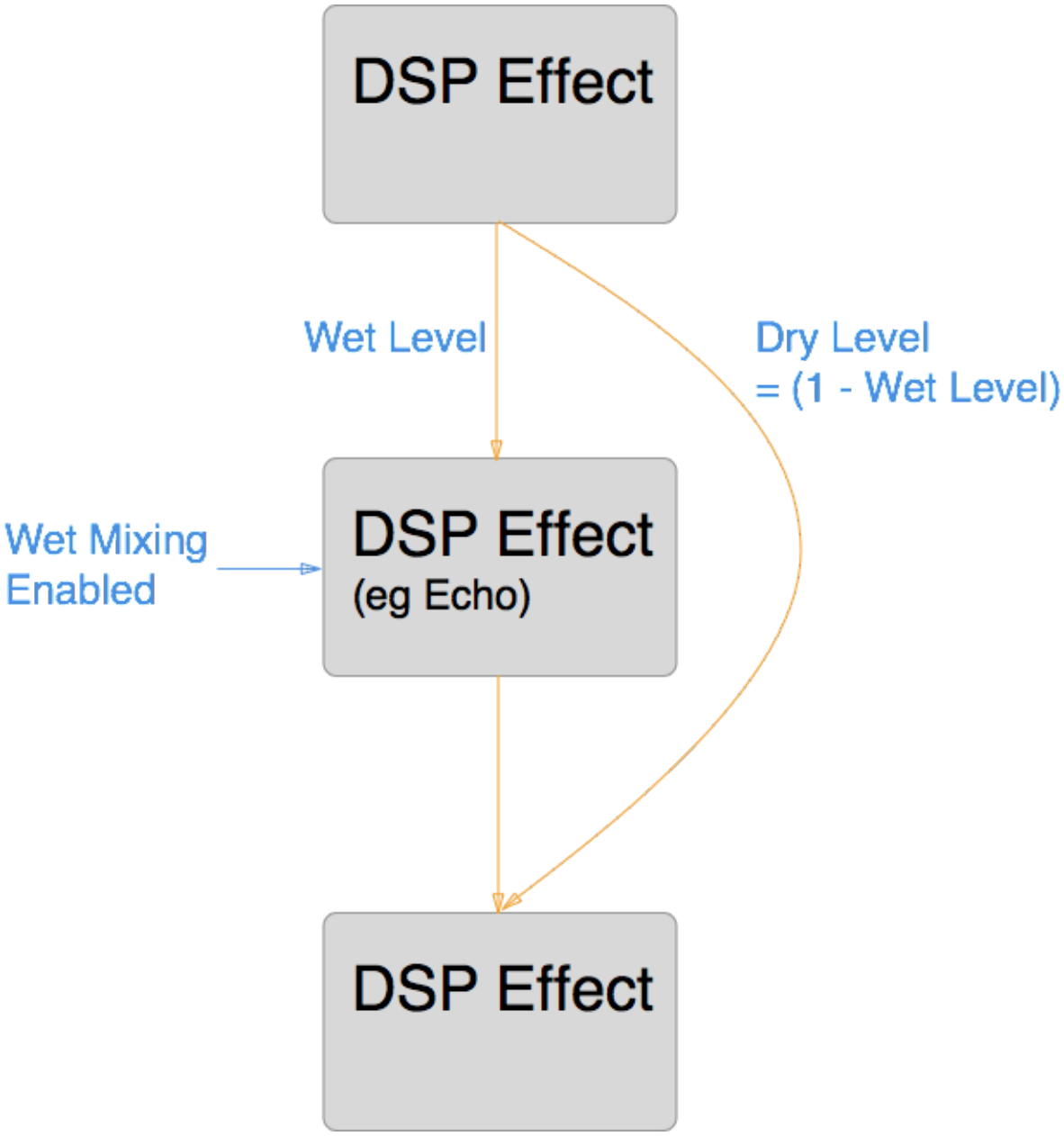
Gear Options

Allow Wet Mixing - Toggling this option creates a dry channel around the effect. The slider that appears when this is enabled dictates what percentage of the signal is passed into the wet/dry **components**. Enabling this increases memory usage and CPU overhead. This is only available on certain units.
Bypass - Toggling this will bypass the Effect Unit completely, effectively disabling it in the signal chain.
Copy Effect Settings to all Snapshots - Selecting this will copy all the parameter values within this Effect Unit to all the other Snapshots in the AudioMixer. This is useful when adding a new Effect Unit, making changes to that Effect Unit and wanting those settings to be the same across all Snapshots.
Add Effect Before - Allows the insertion of an Effect Unit before the current Effect Unit in the AudioGroup. Select the desired effect from the menu shown.
Add Effect After - Allows the insertion of an Effect Unit after the current Effect Unit in the AudioGroup. Select the desired effect from the menu shown.
Remove This Effect - Remove this Effect Unit completely from the AudioMixer. Attenuation Units cannot be removed from AudioGroups.

Wet Mixing

Allowing Wet Mixing on a DSP effect allows you to decide how much of the audio signal that is to enter the effect actually is processed by the effect. Enabling Wet Mixing effectively creates a dry channel around the effect. You can then click on the effect slot and drag left and right to increase or decrease the percentage of audio signal that is passed through the DSP effect unit. The rest of the signal is passed through the dry channel. The following diagram illustrates this concept:

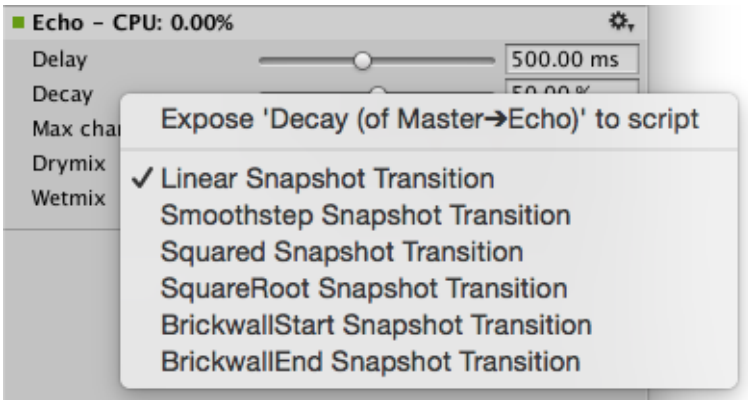
Wet mixing is good for when a user wants to control the influence an effect has on the mix and preserve a percentage of the original signal.



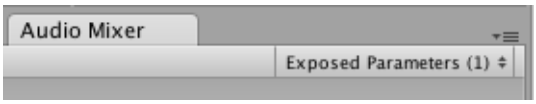
Exposed Parameters

Exposed Parameters allow you to bypass the Snapshot system of an AudioMixer and set the value of any parameter within an AudioMixer from script. When an Exposed Parameter is set via script, that parameter is locked to that value and will not change as the the game transitions Snapshots.

Exposing a parameter with an AudioMixer is done in the AudioGroup Inspector. For any parameter shown in the Inspector (including Pitch, Volume, Send Level and Wet Level), you can right click on the name of the Parameter and choose 'Expose X to script'.



Once a parameter is exposed it will show up in the Exposed Parameter drop down in the top right corner of the AudioMixer Window. Clicking on this drop down will reveal all the Exposed Parameters in the AudioMixer.



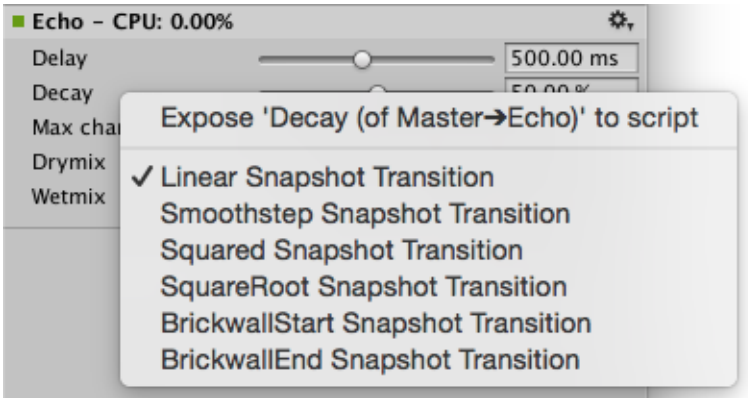
To rename an Exposed Parameter, right click on the name of the Exposed Parameter and click 'Rename'. This name will be how you reference the parameter from the AudioMixer API. *To delete an Exposed Parameter, right click on the name of the Exposed Parameter and click 'Delete'.

Transition Overrides

When transitioning between Snapshots, by default all transitions are done with linear **interpolation** from the beginning to target values. In some cases this transition behaviour is not desired however, for example when it it preferable to brick-wall the change at the start or end of the transition.

All of the parameters available within the AudioMixer can have their transition behaviour changed. Transition behaviours are defined per-Snapshot, with the target Snapshot defining the transition behaviour.

To set the transition override for a particular parameter for the current Snapshot, right click on the parameter name and select the required transition type.

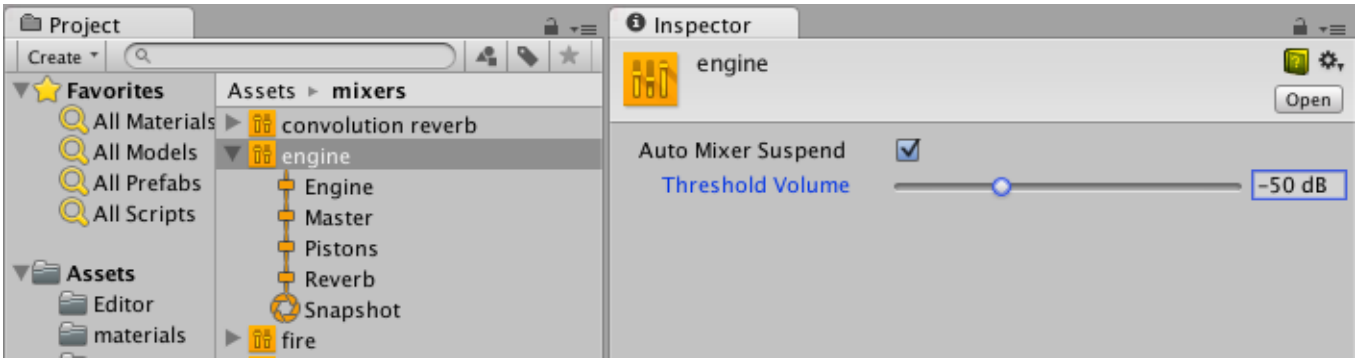


AudioMixer Inspector

The audio mixer **asset** itself has an inspector that allows specifying the overall activation/suspense behavior of the mixer. Being assets, audio mixers are basically activated when any **audio source** plays into the mixer and will stay active as long as there is such a driver supplying audio data to the mixer. Since mixers can also be activated by the audio preview button in the **scene view**, the activation behavior is different from that of scene-objects such as MonoBehaviors. Thus, a mixer may be active (and therefore consuming CPU) even in stop mode.

To avoid running out of CPU resources in a project that contains a large number of mixers that are not supposed to be running all at the same time (say, because specific levels use certain specialized mixers), the audio mixers have functionality built-in to put themselves into suspended mode in which all processing stops. To do this in a natural way that doesn't lead to audible artefacts such as clicks or missing rever/echo tails each mixer uses the following strategy:

As long as any audio source is playing into this mixer or the mixer is receiving audio data from other sub-mixers the mixer will keep itself active. After the last sound source has finished playing, the mixer will wait for a second and then continually use loudness-measurement at its own output do decide if it should suspend itself. This is needed because reverb and echo tails can potentially decay very slowly. The loudness threshold at which the mixer suspends itself is determined by enabling Auto Mixer Suspend and setting the Threshold Volume parameter on the mixer asset's inspector which is shown then the mixer asset is selected in the project browser (not when selecting a sub-asset like a mixer group or snapshot). The value of –80 dB is chosen as the default and matches the lowest value of the faders in the mixer. In practice it is often possible to set it to a significantly larger value to get quicker deactivation and avoid intermediate CPU spikes that could cause stutter.



Suspend settings on audio mixer asset

Overview of Usage and API

Leave feedback

The AudioMixer is a very self-contained **asset** with a streamlined API.

Using the Snapshot and AudioGroup objects
Transitioning Snapshots
Blending Snapshot states
Finding Groups
Re-routing at Runtime

Example Usages

A couple of music-related demos can be found in the audio demos repository.

Also the native audio plugin repository provides the official SDK for developing custom extensions as well as many examples of such involving complex routing and custom processing/generation of audio in the mixer.

Native Audio Plugin SDK

Leave feedback

This document describes the built-in native audio **plugin** interface of Unity 5.0. We will do this by looking at some specific example plugins that grow in complexity as we move along. This way, we start out with very basic concepts and introduce more complex use-cases near the end of the document.

Download

First thing you need to do is to download the newest audio plugin SDK from here.

Overview

The native audio plugin system consists of two parts:

The native DSP (Digital Signal Processing) plugin which has to be implemented as a .dll (Windows) or .dylib (OSX) in C or C++. Unlike **scripts** and because of the high demands on performance this has to be compiled for any platform that you want to support, possibly with platform-specific optimizations.

The GUI which is developed in C#. Note that the GUI is optional, so you always start out plugin development by creating the basic native DSP plugin, and let Unity show a default slider-based **UI** for the parameter descriptions that the **native plugin** exposes. We recommend this approach to bootstrap any project.

Note that you can initially prototype the C# GUI as a .cs file that you just drop into the **Assets**/Editor folder (just like any other editor script). Later on you can move this into a proper Visual Studio project as your code starts to grow and need better modularization and better IDE support. This enables you to compile it into a .dll, making it easier for the user to drop into the project and also in order to protect your code.

Also note that both the native DSP and GUI DLLs can contain multiple plugins and that the binding happens only through the names of the effects in the plugins regardless of what the DLL file is called.

What are all these files?

The native side of the plugin SDK actually only consists of one file (AudioPluginInterface.h), but to make it easy to have multiple plugin effects within the same DLL we have added supporting code to handle the effect definition and parameter registration in a simple unified way (AudioPluginUtil.h and AudioPluginUtil.cpp). Note that the NativePluginDemo project contains a number of example plugins to get you started and show a variety of different plugin types that are useful in a game context. We place this code in the public domain, so feel free to use this code as a starting point for your own creations.

Development of a plugin starts with defining which parameters your plugin should have. You don't need to have a detailed master plan of all the parameters that the plugin will have laid out before you start, but it helps to roughly have an idea of how you want the user experience to be and what **components** you will need.

The example plugins that we provide have a bunch of utility functions that make it easy Let's take a look at the "Ring Modulator" example plugin. This simple plugin multiplies the incoming signal by a sine wave, which gives a nice radio-noise / broken reception like effect, especially if multiple ring modulation effects with different frequencies are chained.

The basic scheme for dealing with parameters in the example plugins is to define them as enum-values that we use as indices into an array of floats for both convenience and brevity.

```
enum Param
{
    P_FREQ,
    P_MIX,
    P_NUM
};

int InternalRegisterEffectDefinition(UnityAudioEffectDefinition& definition)
```

```
{
    int numparams = P_NUM;
    definition.paramdefs = new UnityAudioParameterDefinition [numparams];
    RegisterParameter(definition, "Frequency", "Hz",
        0.0f, kMaxSampleRate, 1000.0f,
        1.0f, 3.0f,
        P_FREQ);
    RegisterParameter(definition, "Mix amount", "%",
        0.0f, 1.0f, 0.5f,
        100.0f, 1.0f,
        P_MIX);
    return numparams;
}
```

The numbers in the RegisterParameter calls are the minimum, maximum and default values followed by a scaling factor used for display only, i.e. in the case of a percentage-value the actual value goes from 0 to 1 and is scaled by 100 when displayed. There is no custom GUI code for this, but as mentioned earlier, Unity will generate a default GUI from these basic parameter definitions. Note that no checks are performed for undefined parameters, so the AudioPluginUtil system expects that all declared enum values (except `P_NUM`) are matched up with a corresponding parameter definition.

Behind the **scenes** the RegisterParameter function fills out an entry in the UnityAudioParameterDefinition array of the UnityAudioEffectDefinition structure that is associated with that plugin (see "AudioEffectPluginInterface.h"). The rest that needs to be set up in UnityAudioEffectDefinition is the callbacks to the functions that handle instantiating the plugin (CreateCallback), setting/getting parameters (`SetFloatParameterCallback/UnityAudioEffect_GetFloatParameterCallback`), doing the actual processing (`UnityAudioEffect_ProcessCallback`) and eventually destroying the plugin instance when done (`UnityAudioEffect_ReleaseCallback`).

To make it easy to have multiple plugins in the same DLL, each plugin is residing in its own namespace, and a specific naming convention for the callback functions is used such that the `DEFINE_EFFECT` and `DECLARE_EFFECT` macros can fill out the UnityAudioEffectDefinition structure. Underneath the hood all the effects definitions are stored in an array to which a pointer is returned by the only entry point of the library UnityGetAudioEffectDefinitions.

This is useful to know in case you want to develop bridge plugins that map from other plugin formats such as VST or AudioUnits to or from the Unity audio plugin interface, in which case you need to develop a more dynamic way to set up the parameter descriptions at load time.

Instantiating the plugin

The next thing is the data for the instance of the plugin. In the example plugins, we put all this into the EffectData structure. The allocation of this must happen in the corresponding CreateCallback which is called for each instance of the plugin in the mixer. In this simple example there's only one sine-wave that is multiplied to all channels, other more advanced plugins need allocate additional data per input channel.

```
struct EffectData
{
    struct Data
    {
        float p[P_NUM]; // Parameters
        float s;        // Sine output of oscillator
        float c;        // Cosine output of oscillator
    };
    union
```

```
    {
        Data data;
        unsigned char pad[(sizeof(Data) + 15) & ~15];
    };
};
```

```
UNITY_AUDIODSP_RESULT UNITY_AUDIODSP_CALLBACK CreateCallback(
    UnityAudioEffectState* state)
{
    EffectData* effectdata = new EffectData;
    memset(effectdata, 0, sizeof(EffectData));
    effectdata->data.c = 1.0f;
    state->effectdata = effectdata;
    InitParametersFromDefinitions(
        InternalRegisterEffectDefinition, effectdata->data.p);
    return UNITY_AUDIODSP_OK;
}
```

The UnityAudioEffectState contains various data from the host such as the sampling rate, the total number of samples processed (for timing), or whether the plugin is bypassed, and is passed to all callback functions.

And obviously to free the plugin instance there is a corresponding function too:

```
UNITY_AUDIODSP_RESULT UNITY_AUDIODSP_CALLBACK ReleaseCallback(
    UnityAudioEffectState* state)
{
    EffectData::Data* data = &state->GetEffectData<EffectData>()->data;
    delete data;
    return UNITY_AUDIODSP_OK;
}
```

The main processing of audio happens in the ProcessCallback:

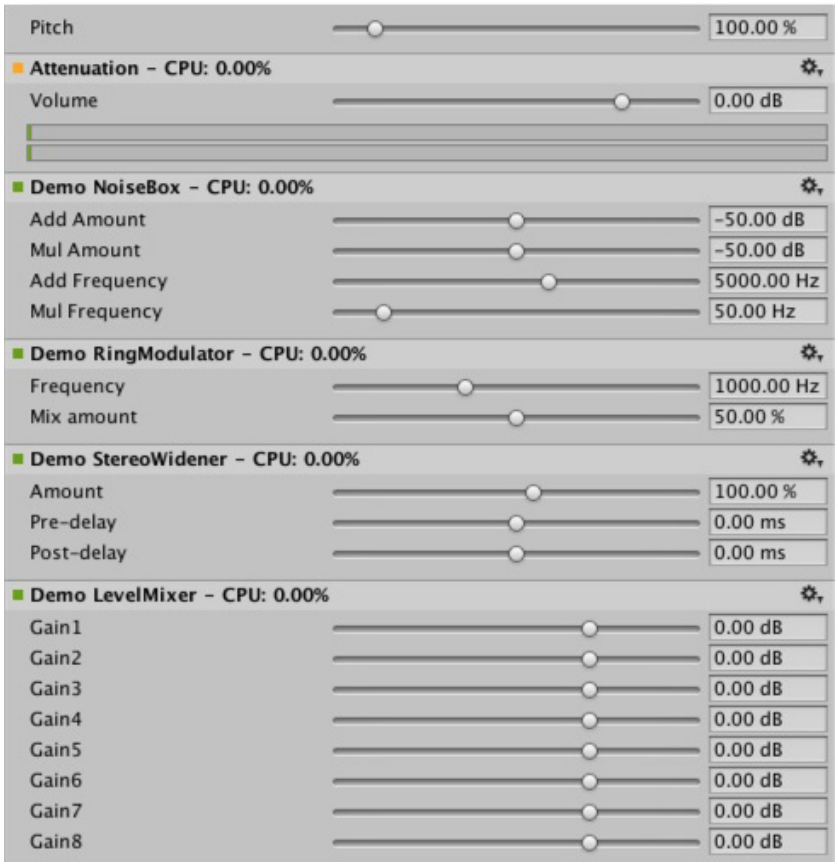
```
UNITY_AUDIODSP_RESULT UNITY_AUDIODSP_CALLBACK ProcessCallback(
    UnityAudioEffectState* state,
    float* inbuffer, float* outbuffer,
    unsigned int length,
    int inchannels, int outchannels)
{
    EffectData::Data* data = &state->GetEffectData<EffectData>()->data;

    float w = 2.0f * sinf(kPI * data->p[P_FREQ] / state->samplerate);
    for(unsigned int n = 0; n < length; n++)
    {
        for(int i = 0; i < outchannels; i++)
```

```
        {
            outbuffer[n * outchannels + i] =
                inbuffer[n * outchannels + i] *
                (1.0f - data->p[P_MIX] + data->p[P_MIX] * data->s);
        }
        data->s += data->c * w; // cheap way to calculate a sine-wave
        data->c -= data->s * w;
    }

    return UNITY_AUDIODSP_OK;
}
```

The GetEffectData function at the top is just a helper function casting the effectdata field of the state variable to the EffectData::Data in the structure we declared above.

Other simple plugins included are the NoiseBox plugin, which adds and multiplies the input signal by white noise at variable frequencies, or the Lofinator plugin, which does simple downsampling and quantization of the signal. All of these may be used in combination and with game-driven animated parameters to simulate anything from mobile phones to bad radio reception on walkies, broken loudspeakers **etc**.

The StereoWidener, which decomposes a stereo input signal into mono and side components with variable delay and then recombines these to increase the perceived stereo effect.



A bunch of simple plugins without custom GUIs to get started with.

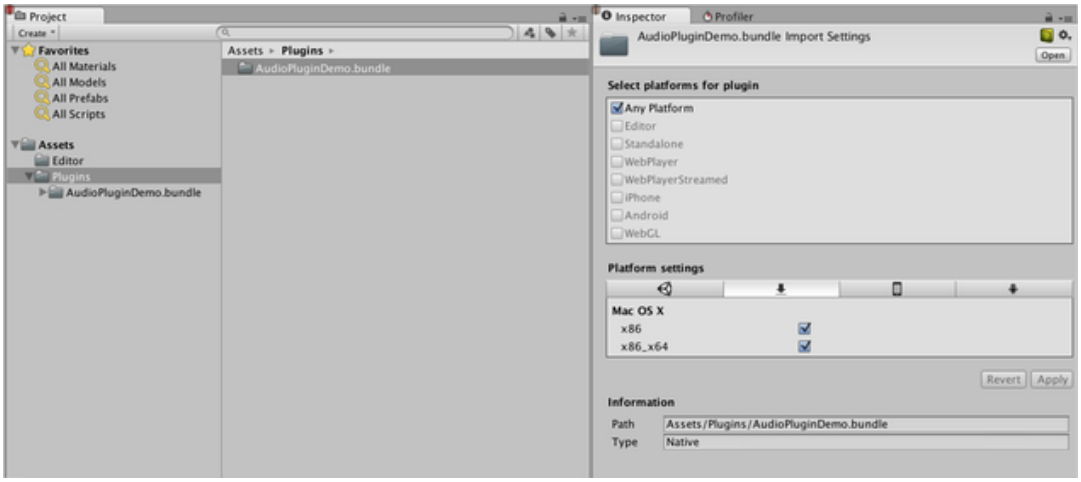
Which plugin to load on which platform?

Native audio plugins use the same scheme as other native or **managed plugins** in that they must be associated with their respective platforms via the plugin importer **inspector**. You can read more about the subfolders in which to place plugins here. The platform association is necessary so that the system knows which plugins to include on a each build target in the standalone builds, and with

the introduction of 64-bit support this even has to be specified within a platform. OSX plugins are special in this regard since the Universal Binary format allows them to contain both 32 and 64 bit variants in the same bundle.

Native plugins in Unity that are called from managed code get loaded via the [DllImport] attribute referencing the function to be imported from the native DLL. However, in the case of native audio plugins things are different. The special problem that arises here is that the audio plugins need to be loaded before Unity starts creating any mixer assets that may need effects from the plugins. In the editor this is no problem, because we can just reload and rebuild the mixers that depend on plugins, but in standalone builds the plugins must be loaded before we create the mixer assets. To solve this, the current convention is to prefix the DLL of the plugin "audioplugin" (case insensitive) so that the system can detect this and add it to a list of plugins that will automatically be loaded at start. Remember that it's only the definitions inside the plugin that define the names of the effects that show up inside Unity's mixer, so the DLL can be called anything, but it needs to start with the string "audioplugin" to be detected as such.

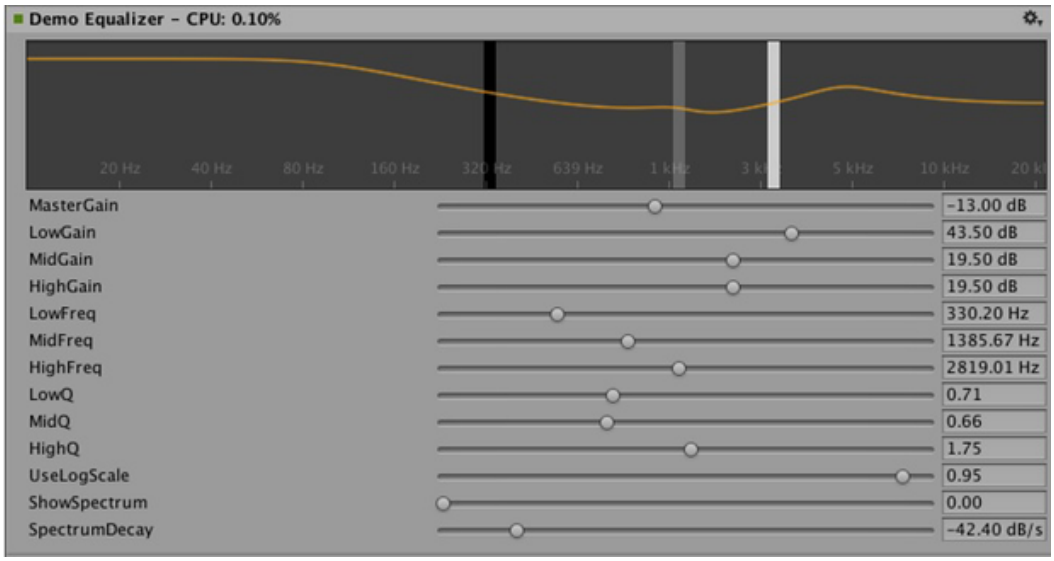
For platforms such as **IOS** the plugin code needs to be statically linked into the Unity binary produced by the generated XCode project and there - just like plugin **rendering** devices - the plugin registration has to be added explicitly to the startup code of the app.



On OSX one bundle can contain both the 32- and 64 bit version of the plugin. You can also split them to save size.

Plugins with custom GUIs

Now let's look at something a little more advanced: Effects for equalization and multiband **compression**. Such plugins have a much higher number of parameters than the simple plugins presented in the previous section and also there is some physical coupling between parameters that require a better way to visualize the parameters than just a bunch of simple sliders. Consider an equalizer for instance: Each band has 3 different filters that collectively contribute to the final equalization curve and each of these filters has the 3 parameters frequency, Q-factor and gain which are physically linked and define the shape of each filter. So it helps the user a lot, if an equalizer plugin has a nice big display showing the resulting curve, the individual filter contributions and can be operated in such a way that multiple parameters can be set simultaneously by simple dragging operations on the control instead of changing sliders one at a time.



Custom GUI of the Equalizer plugin. Drag the three bands to change the gains and frequencies of the filter curve. Hold shift down while dragging to change the shape of each band.

So once again, the definition, initialization, deinitialization and parameter handling follows the exact same enum-based method that the simple plugins use, and even the ProcessCallback code is rather short. Well, time to stop looking at the native code and open the AudioPluginDemoGUI.sln project in Visual Studio. Here you will find the associated C# classes for the GUI code. The way it works is simple: Once Unity has loaded the native plugin DLLs and registered the contained audio plugins, it will start looking for corresponding GUIs that match the names of the registered plugins. This happens through the Name property of the EqualizerCustomGUI class which, like all custom plugin GUIs, must inherit from IAudioEffectPluginGUI. There's only one important function inside this class which is the bool OnGUI(IAudioEffectPlugin plugin) function. Via the IAudioEffectPlugin plugin argument this function gets a handle to the native plugin that it can use to read and write the parameters that the native plugin has defined. So to read a parameter it calls:

```
plugin.GetFloatParameter("MasterGain", out masterGain);
```

which returns true if the parameter was found, and to set it, it calls:

```
plugin.SetFloatParameter("MasterGain", masterGain);
```

which also returns true if the parameter exists. And that's basically the most important binding between GUI and native code. You can also use the function

```
plugin.GetFloatParameterInfo("NAME", out minVal, out maxVal, out defVal);
```

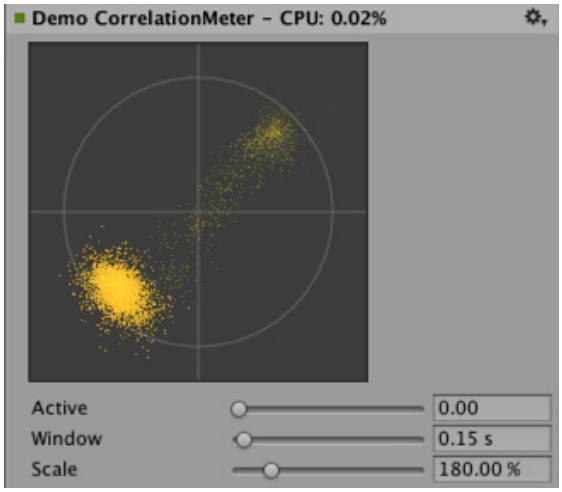
to query parameter "NAME" for it's minimum, maximum and default values to avoid duplicate definitions of these in the native and UI code. Note that if your OnGUI function return true, the Inspector will show the default UI sliders below the custom GUI. This is again useful to bootstrap your GUI development as you have all the parameters available while developing your custom GUI and have an easy way to check that the right actions performed on it result in the expected parameter changes.

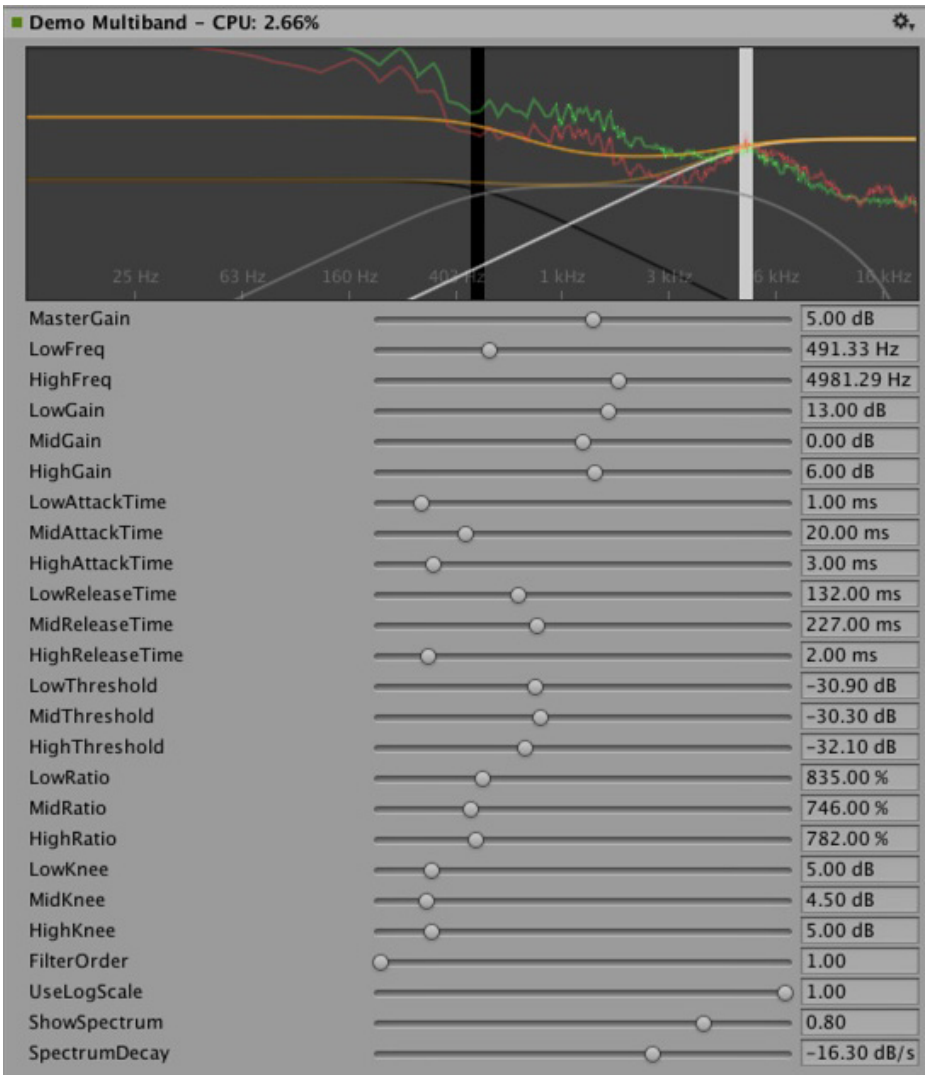
We won't discuss the details about the DSP processing that is going on in the Equalizer and Multiband plugins here, for those interested, the filters are taken from Robert Bristow Johnson's excellent Audio EQ Cookbook and to plot the curves Unity provides some internal API functions to draw antialiased curves for the frequency response.

One more thing to mention though is that both the Equalizer and Multiband plugins do also provide code to overlay the input and output spectra to visualize the effect of the plugins, which brings up an interesting point: The GUI code runs at much lower update rate (the frame rate) than the audio processing and doesn't have access to the audio streams, so how do we read this data? For this, there is a special function for this in the native code:

```
UNITY_AUDIODSP_RESULT UNITY_AUDIODSP_CALLBACK GetFloatParameterCallback(
    UnityAudioEffectState* state,
    int index,
    float* value,
    char *valuestr)
{
    EffectData::Data* data = &state->GetEffectData<EffectData>()->data;
    if(index >= P_NUM)
        return UNITY_AUDIODSP_ERR_UNSUPPORTED;
    if(value != NULL)
        *value = data->p[index];
    if(valuestr != NULL)
        valuestr[0] = 0;
  return UNITY_AUDIODSP_OK;
}
```

It simply enables reading an array of floating-point data from the native plugin. Whatever that data is, the plugin system doesn't care about, as long as the request doesn't massively slow down the UI or the native code. For the Equalizer and Multiband code there is a utility class called FFTAnalyzer which makes it easy to feed in input and output data from the plugin and get a spectrum back. This spectrum data is then resampled by GetFloatBufferCallback and handed to the C# UI code. The reason that the data needs to be resampled is that the FFTAnalyzer runs the analysis at a fixed frequency resolution while GetFloatBufferCallback just returns the number of samples requested, which is determined by the width of the view that is displaying the data. For a very simple plugin that has a minimal amount of DSP code you might also take a look at the CorrelationMeter plugin, which simply plots the amplitude of the left channel against the amplitude of the right channel in order to show "how stereo" the signal is.

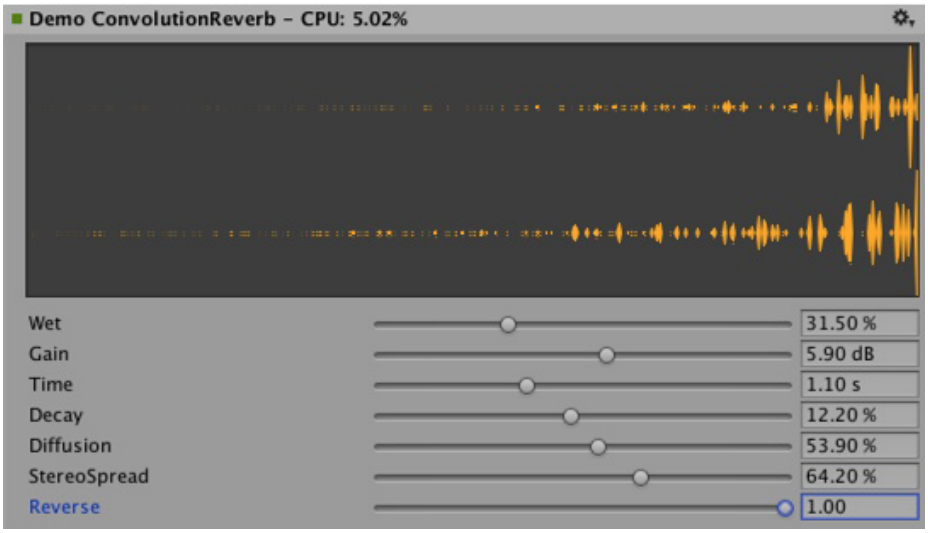




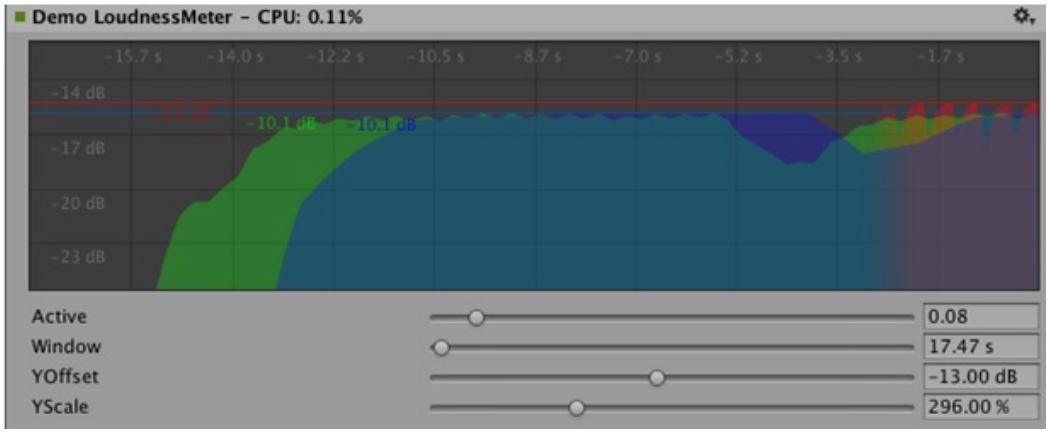
Left: Custom GUI of the CorrelationMeter plugin.

Right: Equalizer GUI with overlaid spectrum analysis (green curve is source, red is processed).

At this point we would also like to point out that both the Equalizer and Multiband effects are kept intentionally simple and unoptimized, but we think they serve as good examples of more complex UIs that are supported by the plugin system. There's obviously a lot of work still in doing the relevant platform-specific optimizations, tons of parameter tweaking to make it fell really right and respond in the most musical way etc. etc... We might also implement some of these effects as built-in plugins in Unity at some point simply for the convenience of increasing Unity's standard repertoire of plugins, but we sincerely hope that the reader will also take up the challenge to make some really awesome plugins – and who knows, they might at some point end up as built-in plugins. ;-)



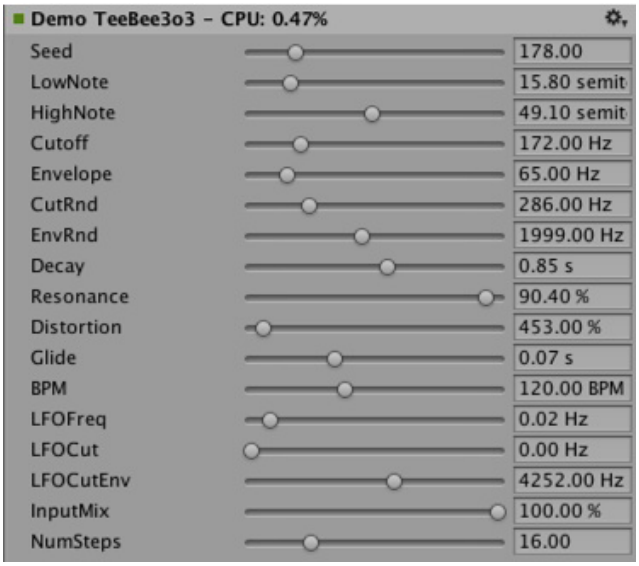
Convolution reverb example plugin. The impulse response is decaying random noise, defined by the parameters. This is only for demonstration purposes, as a production plugin should allow the user to load arbitrary recorded impulses, the underlying convolution algorithm remains the same nevertheless.

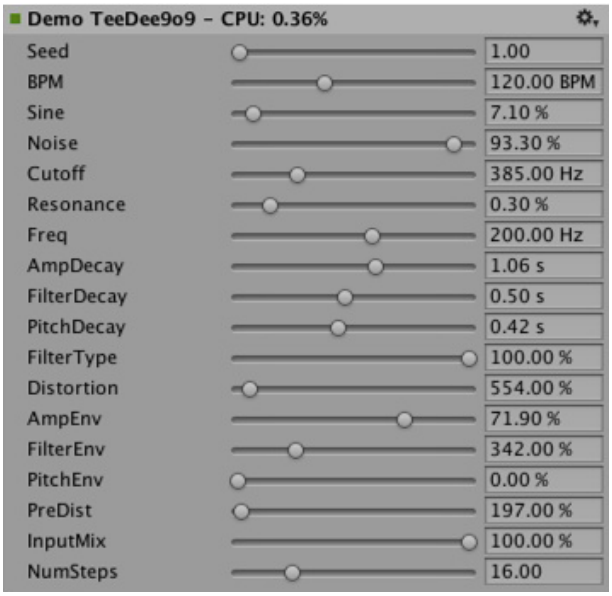


Example of a loudness monitoring tool measuring levels at 3 different time scales. Also just for demonstration purposes, but a good place to start building a monitor tool that conforms to modern loudness standardizations. The curve rendering code is built into Unity.

Synchronizing to the DSP clock

Time for some fun exercises. Why not use the plugin system to generate sound instead of just processing it? Let's try to do some simple bassline and drum synthesizers that should be familiar to people who listen to acid trance – some simple clones of some of the main synths that defined this genre. Take a look at Plugin_TeeBee.cpp and Plugin_TeeDee.cpp. These simple synths just generate patterns with random notes and have some parameters for tweaking the filters, envelopes and so forth in the synthesis engine. Again, we won't discuss those details here, but just point out that the state->dsptick parameter is read in the ProcessCallback in order to determine the position in the "song". This counter is a global sample position, so we just divide it by the length of each note specified in samples and fire a note event to the synthesis engine whenever this division has a zero remainder. This way, all plugin effects stay in sync to the same sample-based clock, and if you would for instance play a prerecorded piece of music with a known tempo through such an effect, you could use the timing info to apply tempo-synchronized filter effects or delays on the music.





Simple bassline and drum synthesizers to demonstrate tempo-synchronized effects.

Spatialization

The native audio plugin SDK is the foundation of the Spatialization SDK which allows developing custom spatialization effects that are instantiazed per **audio source**. More information about this can be found here.

Outlook

This is just the start of an effort to open up parts of the sound system to high performance native code. We have plans to integrate this in other parts of Unity as well to make the effects usable outside of the mixer as well as extending the SDK to support other parameter types than floats with support for better default GUIs as well as storage of binary data.

Have a lot of fun creating your own plugins. Hope to see them on the **asset store**. ;-)

“Disclaimer”

While there are many similarities in the design, Unity’s native audio SDK is not built on top of other plugin SDKs like Steinberg VST or Apple AudioUnits. It should be mentioned that it would be easy for the interested reader to implement basic wrappers for these using this SDK that allow using such plugins to be used in Unity. It is not something the dev-team of Unity is planning to do. Proper hosting of any plugin quickly gets very complex, and dealing with all the intricacies of expected invocation orders and handling custom GUI windows that are based on native code quickly grows by leaps and bounds which makes it less useful as example code.

While we do understand that it could potentially be quite useful to load your VST or AU plugin or even effects for just mocking up / testing sound design, bear in mind that using VST/AU also limits you to a few specific platforms. The potential of writing audio plugins based on the Unity SDK is that it extends to all platforms that support software-mixing and dynamically loaded native code. That said, there are valid use cases for mocking up early sound design with your favourite tools before deciding to devote time to develop custom plugins (or simply to be able to use metering plugins in the editor that don't alter the sound in any way), so if anyone wants to make a nice solution for that, please do.

2018–03–19 Page amended with limited editorial review

MonoDevelop replaced by Visual Studio from 2018.1

Audio Spatializer SDK

Leave feedback

Overview

The **audio spatializer** SDK is an extension of the native audio **plugin** SDK that allows changing the way audio is transmitted from an audio source into the surrounding space. The built-in panning of **audio sources** may be regarded as a simple form of spatialization in that it takes the source and regulates the gains of the left and right ear contributions based on the distance and angle between the AudioListener and the AudioSource. This provides simple directional cues for the player on the horizontal plane.

Background

With the advent of virtual and augmented reality systems the spatialization method more and more becomes a key **component** of the player's immersion. Our ears and brains are highly aware of microscopic delays between the sound received from a source at the left and right ears respectively. Furthermore we are capable of unconsciously interpreting a change in the balance of high frequencies to tell if an object is in front of, behind or even above or below us. We may also be able to tell if an object is partially occluded based on the difference in the sound at each ear or infer something about the shape of the room that we are in based on the reflections of the sound. In other words: Sound is extremely important in our daily navigation, we just maybe don't notice it that much!

Sound occlusion is a very hard problem to solve in terms of computation power. Whereas in global illumination you may consider the movement of light as effectively instantaneous, sound is moving very slowly. Therefore calculating the way sound actually moves around (as waves) in a room is not feasible computationally. For the same reason there are many approaches towards spatialisation tackling different problems to various extents.

Some solutions are only solving the HRTF problem. HRTF stands for Head-Related Transfer Function, and a rough analogy to this from the graphics world would be spherical harmonics: i.e. a directionally influenced filtering of the sound that we apply on both ears which contains the micro-delay between the ears as well as the directional filtering that ear-flaps, the head itself and the shoulders contribute to. Adding HRTF filtering already immensely improves the sensation of direction over a conventional panning solution (a typical and famous example of this is the binaural recording of the virtual barber shop). Direct HRTF is somewhat limited though as it only is concerned with the direct path of audio and not how it is transmitted in space.

Occlusion is the next step up from this in that it can indirectly reflect the sound off walls. To take a rough equivalent from the graphics world again, this could be compared to specular reflection in the sense that both source and listener locations determine the outcome, and of course each reflected directional wave of sound hits each ear with a different HRTF and has a different delay based on the length of the path that the wave has travelled.

Finally there is room reflections which in many ways corresponds to the diffuse part of a global illumination solution in that sound gets emitted into the room and is reflected on multiple walls before hitting the ears as a field of overlapping waves, each with a different direction and accumulated delay relative to the audio source.

SDK and Example Implementation

With so many hard problems to solve there exist a variety of different audio spatialization solutions. We found that the best way to support these in Unity was to create an open interface, the Audio Spatializer SDK, which is an extension on top of the Native Audio Plugin SDK that allows replacing the standard panner in Unity by a more advanced one and gives access to important meta-data about the source and listener needed for the computation.

An example implementation of a spatializer is provided here. It is intentionally simple in that it only supports direct HRTF and needs to be optimized for production use. Accompanying the plugin is a simple reverb, just to show how audio data can be routed from the spatializer plugin to the reverb plugin. The HRTF filtering is based on the KEMAR data set, which is a set of per-ear impulse response recordings performed on a dummy head by Bill Gardner at MIT Media Lab. These impulse responses are convolved with the input signal using fast convolution via the Fast Fourier Transform. The positional meta-data is only used for picking the right impulse response sets, as the data set consists of circularly arranged impulse responses for elevation angles ranging from –40 below to 90 degrees above the head.

Initialization

The main difference between a spatialization effect and mixer effects in Unity is that the spatializer is placed right after the audio source decoder that produces a stream of audio data so that each source has its own effect instance processing only the audio produced by the source. This is different from audio mixer plugins that process a mixture of audio from various audio sources connected to a mixer group. To enable the plugin to operate like this it is necessary to set a flag in the description bit-field of the effect:

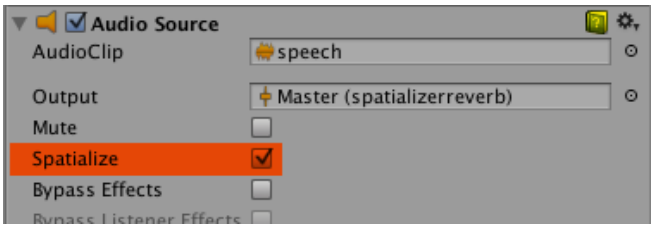
```
definition.flags |= UnityAudioEffectDefinitionFlags_IsSpatializer;
```

Setting this flag upon initialization notifies Unity during the plugin scanning phase that this is a spatializer and so, when an instance of this plugin is created, will allocate the UnityAudioSpatializerData structure for the spatializerdata member of the UnityAudioEffectState structure.

Before being able to use the spatializer in the project, it needs to be selected in the Audio **Project settings**:

Spatializer plugin selector

On the AudioSource, the checkbox Spatialize enables the spatializer to be used. This may also be controlled from script via the AudioSource.spatialize property. In a game with a lot of sounds it may make sense to only enable the spatializer on the nearby sounds and use traditional panning on the distant ones.



Spatializer checkbox on AudioSource

Spatializer effect meta-data

Unlike other effects that are run in the mixer on a mixture of sounds, spatializers are applied directly after the AudioSource has decoded audio data. Therefore each instance of the spatializer effect has an own instance of UnityAudioSpatializerData associated with mainly data about the AudioSource.

```
struct UnityAudioSpatializerData
{
    float listenermatrix[16]; // Matrix that transforms sourcepos into the local space of the
    float sourcematrix[16];   // Transform matrix of audio source
    float spatialblend;       // Distance-controlled spatial blend
    float reverbzonemix;      // Reverb zone mix level parameter (and curve) on audio source
    float spread;             // Spread parameter of the audio source (0..360 degrees)
```

```
    float stereopan;          // Stereo panning parameter of the audio source (-1: fully left,
                              // The spatializer plugin may override the distance attenuation
                              // influence the voice prioritization (leave this callback as NU
                              // built-in audio source attenuation curve)
    UnityAudioEffect_DistanceAttenuationCallback distanceattenuationcallback;
};
```

The structure contains the full 4x4 transform matrices for the listener and source. The listener matrix has already been inverted so that the two matrices can be easily multiplied to get a relative direction-vector. The listener matrix is always orthonormal, so the inverse is cheap to calculate. Furthermore the structure contains fields corresponding to the properties of the audio source: Spatial Blend, Reverb Zone Mix, Spread and Stereo Pan. It is the responsibility of the spatializer to implement these correctly, as when it's active, Unity's audio system will only provide the raw source sound as a stereo signal (even when the source is mono or multi-channel in which case up- or down-mixing is used).

Matrix conventions

The sourcematrix field contains a plain copy of the transformation matrix associated with the AudioSource . For a plain AudioSource on a game object that is not rotated that will just be a translation matrix where the position is encoded in elements 12, 13 and 14. The listenermatrix field contains the inverse of the transform matrix associated with the AudioListener. This makes it very convenient to determine the direction vector from the listener to the source like this:

```
float dir_x = L[0] * S[12] + L[4] * S[13] + L[ 8] * S[14] + L[12];
float dir_y = L[1] * S[12] + L[5] * S[13] + L[ 9] * S[14] + L[13];
float dir_z = L[2] * S[12] + L[6] * S[13] + L[10] * S[14] + L[14];
```

where L is listenermatrix and S is sourcematrix. If you have a listenermatrix that is not rotated and has uniform scaling of 1 (camera matrices should never be scaled), notice that the position in (L[12], L[13], L[14]) is actually the negative value of what you see in Unity's **inspector**. This is because listenermatrix is the inverse of the **camera**'s transformation matrix. If the **camera** had also been rotated we would not be able to read the positions directly from the matrix simply by negating, but would have to undo the effect of the rotation first. Luckily it is easy to invert such Transformation-Rotation-Scaling matrices as described here, so what we need to do is transpose the top-left 3x3 rotation matrix of L and calculate the positions like this:

```
float listenerpos_x = -(L[0] * L[12] + L[ 1] * L[13] + L[ 2] * L[14]);
float listenerpos_y = -(L[4] * L[12] + L[ 5] * L[13] + L[ 6] * L[14]);
float listenerpos_z = -(L[8] * L[12] + L[ 9] * L[13] + L[10] * L[14]);
```

Attenuation curves and audibility

The only thing that is still handled by the Unity audio system is the distance attenuation, which is applied to the sound before it enters the spatialization stage, and this is necessary so that the audio system knows the approximate audibility of the source, which can be used for dynamic virtualization of sounds based on importance in order to match the user-defined Max Real Voices limit. Since this is a chicken-and-egg problem this information is not retrieved from actual signal level measurements but corresponds to the combination of the values that we read from the distance-controlled attenuation curve, the Volume property and attenuations applied by the mixer. It is however possible to override the attenuation-curve by your own or use the value calculated by the AudioSource's curve as a base for modification. To do this there is a callback in the UnityAudioSpatializerData structure that may be implemented:

```
typedef UNITY_AUDIODSP_RESULT (UNITY_AUDIODSP_CALLBACK* UnityAudioEffect_DistanceAttenuationCa
    UnityAudioEffectState* state,
    float distanceIn,
    float attenuationIn,
    float* attenuationOut);
```

A simple custom logarithmic curve may just be implemented like this:

```
UNITY_AUDIODSP_RESULT UNITY_AUDIODSP_CALLBACK SimpleLogAttenuation(
    UnityAudioEffectState* state,
    float distanceIn,
    float attenuationIn,
    float* attenuationOut)
{
    const float rollOffScale = 1.0f; // Similar to the one in the Audio Project Settings
    *attenuationOut = 1.0f / max(1.0f, rollOffScale * distanceIn);
    return UNITY_AUDIODSP_OK;
}
```

Script API

Complementing the native side there are also two new methods on the AudioSource that allow setting and getting parameters from the spatializer effect. These are named SetSpatializerFloat/GetSpatializerFloat and work similarly to the SetFloatParameter/GetFloatParameter used in the generic native audio plugin interface. The main difference is that SetSpatializerFloat/GetSpatializerFloat take and index to the parameter to be set/read whereas SetSpatializerFloat/GetSpatializerFloat refer to the parameters by name.

Additionally the boolean property AudioSource.spatializer is linked to the checkbox in the AudioSource inspector and controls the instantiation and deallocation of the spatializer effect based on the selection in the Audio Project Settings. If instantiation of your spatializer effect is a very costly thing (in terms of memory allocations, precalculations etc) you may consider keeping the Unity plugin interface bindings very light-weight and dynamically allocate your effects from a pool so that the activation/deactivation does not lead to frame drops.

Known limitations of the example plugin

Due to the fast convolution algorithm used, moving fast causes some zipper artefacts which can be removed though the use of overlap-save convolution or cross-fading buffers. Also the code does not support tilting the head to the side, this should be easy to fix though. The KEMAR data set is the only data set used in this demo. IRCAM has a few data sets available that were obtained from human subjects.

Audio Profiler

Leave feedback

The **profiler** window contains an audio pane that can reveal detailed information about performance metrics and a log for any audio activity for the past rendered frames. See Audio Area for more information.

Ambisonic Audio

Leave feedback

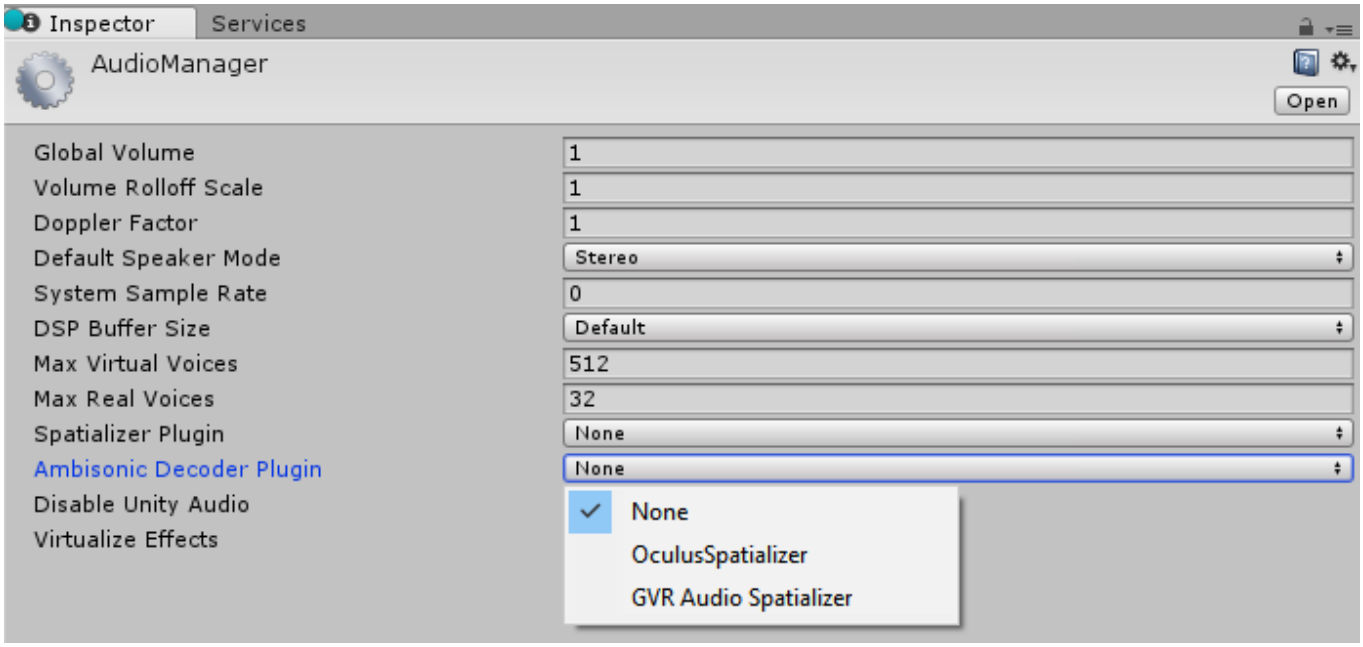
Introduction

This page explains how to play back ambisonics and the changes to our audio **plugin** interface to support ambisonic audio decoders.

Ambisonics are stored in a multi-channel format. Instead of each channel being mapped to a specific speaker, ambisonics instead represent the soundfield in a more general way. The soundfield can then be rotated based on the listener's orientation (i.e. the user's head rotation in XR). The soundfield can also be decoded into a format that matches the speaker setup. Ambisonics are commonly paired with 360-degree videos and can also be used as an audio **skybox**, for distant ambient sounds.

Selecting an ambisonic audio decoder

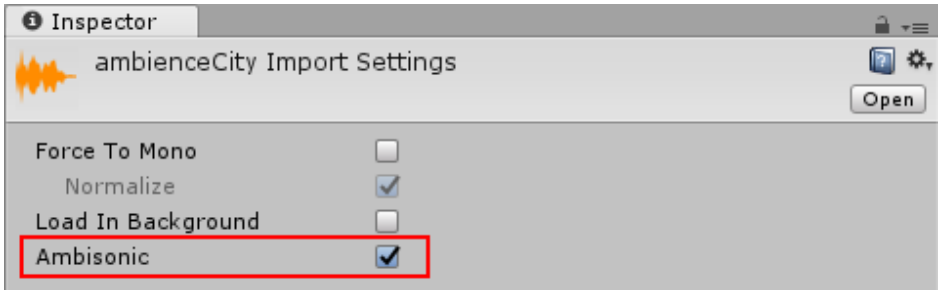
Navigate to the project's Audio Manager inspector window via **Edit > Project Settings > Audio**. Select an ambisonic decoder from the list of available decoders in the project. There are no built-in decoders that ship with Unity 2017.1, but Google and Oculus will each provide one in their audio SDKs for Unity.



Ambisonic options in the Audio Settings

Import an ambisonic audio clip

Import a multi-channel WAV file just like normal. In the **audio clip**'s **inspector** window, select the new ambisonic check-box. The WAV file should be B-format, with ACN **component** ordering, and SN3D normalization.



The Ambisonic checkbox in the audio clip inspector

Play the ambisonic audio clip through an audio source

Play the ambisonic audio clip through an **audio source** just like you would any other audio clip. When an ambisonic clip is played, it will first be de-compressed if necessary, then sent through an ambisonic decoder to convert it to the project's selected speaker mode, then sent through the __audio source__'s effects.

There are a couple of things to note related to audio source properties:

Set spatialize to false. When an ambisonic audio clip is played, it is automatically sent through the project's selected ambisonic audio decoder. The decoder converts the clip from ambisonic format to the project's selected speaker format. It also already handles spatialization as a part of this decoding operation, based on the orientation of the audio source and **audio listener**.

Reverb zones are disabled for ambisonic audio clips, just like they are for spatialized audio sources.

Audio plugin interface changes

For plugin authors, please start by reading about the native audio plugin SDK and **audio spatializer** SDKs in the Unity manual and downloading the audio plugin SDK:

https://docs.unity3d.com/Manual/AudioMixerNativeAudioPlugin.html

https://docs.unity3d.com/Manual/AudioSpatializerSDK.html

https://bitbucket.org/Unity-Technologies/nativeaudioplugins

There are two changes to AudioPluginInterface.h for ambisonic audio decoders. First, there is a new effect definition flag: UnityAudioEffectDefinitionFlags_IsAmbisonicDecoder. Ambisonic decoders should set this flag in the definition bit-field of the effect. During the plugin scanning phase, this flag notifies Unity that this effect is an ambisonic decoder and it will then show up as an option in the Audio Manager's list of ambisonic decoders.

Second, there is a new UnityAudioAmbisonicData struct that is passed into ambisonic decoders, which is very similar to the UnityAudioSpatializerData struct that is passed into spatializers, but there is a new ambisonicOutChannels integer that has been added. This field will be set to the DefaultSpeakerMode's channel count. Ambisonic decoders are placed very early in the audio pipeline, where we are running at the clip's channel count, so ambisonicOutChannels tells the plugin how many of the output channels will actually be used.

If we are playing back a first order ambisonic audio clip (4 channels) and our speaker mode is stereo (2 channels), an ambisonic decoder's process callback will be passed in 4 for the in and out channel count. The ambisonicOutChannels field will be set to 2. In this common scenario, the plugin should output its spatialized data to the first 2 channels and zero out the other 2 channels.

The Unity ambisonic sources framework in 2017.1 can support first and second order ambisonics. The plugin interface includes the information to support binaural stereo, **quad**, 5.1, and 7.1 output, but the level of support is really determined by the plugin. Initially, it is only expected that ambisonic decoder plugins will support 1st order ambisonic sources and binaural stereo output.

There is nothing in the current framework that is specific to any of the different ambisonic formats available. If the clip's format matches the ambisonic decoder plugin's expected format, then everything should just work. Our current plan, though, is that Unity's preferred ambisonic format will be B-format, with ACN component ordering, and SN3D normalization.

2017–08–10 Page published with no editorial review

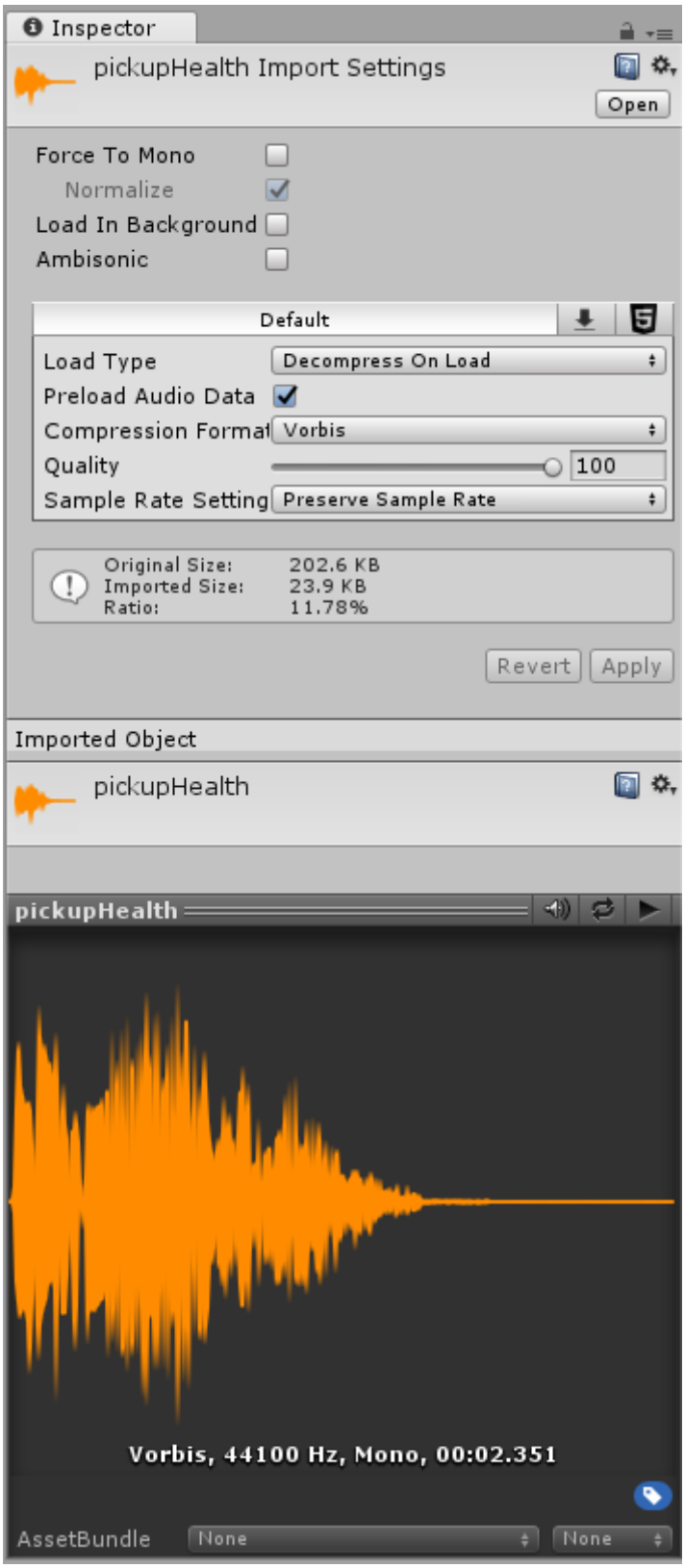
New feature in Unity 2017.1

Audio Clip

Leave feedback

SWITCH TO SCRIPTING

Audio Clips contain the audio data used by Audio Sources. Unity supports mono, stereo and multichannel audio **assets** (up to eight channels). The audio file formats that Unity can import are **.aif**, **.wav**, **.mp3**, and **.ogg**. Unity can also import tracker modules in the **.xm**, **.mod**, **.it**, and **.s3m** formats. The tracker module assets behave the same way as any other audio assets in Unity although no waveform preview is available in the asset import **inspector**.



The Audio Clip inspector

Options

Force To Mono

When this option is enabled, multi-channel audio will be mixed down to a mono track before packing.

Normalize

When this option is enabled, audio will be normalized during the “Force To Mono” mixing down process.

Load In Background

When this option is enabled, the loading of the clip will happen at a delayed time on a separate thread, without blocking the main thread.

Ambisonic

Ambisonic audio sources store audio in a format which represents a soundfield that can be rotated based on the listener’s orientation. It is useful for 360-degree videos and **XR** applications. Enable this option if your audio file contains Ambisonic-encoded audio.

Properties

Property:	*Function:*
Load Type	The method Unity uses to load audio assets at runtime.
Decompress On Load	Audio files will be decompressed as soon as they are loaded. Use this option for smaller compressed sounds to avoid the performance overhead of decompressing on the fly. Be aware that decompressing Vorbis-encoded sounds on load will use about ten times more memory than keeping them compressed (for ADPCM encoding it’s about 3.5 times), so don’t use this option for large files.
Compressed In Memory	Keep sounds compressed in memory and decompress while playing. This option has a slight performance overhead (especially for Ogg/Vorbis compressed files) so only use it for bigger files where decompression on load would use a prohibitive amount of memory. The decompression is happening on the mixer thread and can be monitored in the “DSP CPU” section in the audio pane of the **profiler** window.
Streaming	Decode sounds on the fly. This method uses a minimal amount of memory to buffer compressed data that is incrementally read from the disk and decoded on the fly. Note that decompression happens on the separate streaming thread whose CPU usage can be monitored in the “Streaming CPU” section in the audio pane of the profiler window. **Note**: Streaming clips has an overload of approximately 200KB, even if none of the audio data is loaded.
Compression Format	The specific format that will be used for the sound at runtime. Note that the options available depend on the currently selected build target.
PCM	This option offers higher quality at the expense of larger file size and is best for very short sound effects.
ADPCM	This format is useful for sounds that contain a fair bit of noise and need to be played in large quantities, such as footsteps, impacts, weapons. The **compression** ratio is 3.5 times smaller than PCM, but CPU usage is much lower than the MP3/Vorbis formats which makes it the preferrable choice for the aforementioned categories of sounds.

Property:	*Function:*
Vorbis/MP3	The compression results in smaller files but with somewhat lower quality compared to PCM audio. The amount of compression is configurable via the Quality slider. This format is best for medium length sound effects and music.
Sample Rate Setting	PCM and ADPCM compression formats allow automatically optimized or manual sample rate reduction.
Preserve Sample Rate	This setting keeps the sample rate unmodified (default).
Optimize Sample Rate	This setting automatically optimizes the sample rate according to the highest frequency content analyzed.
Override Sample Rate	This setting allows manual overriding of the sample rate, so effectively this may be used to discard frequency content.
Force To Mono	If enabled, the audio clip will be down-mixed to a single channel sound. After the down-mixing the signal is peak-normalized, because the down-mixing process typically results in signals that are more quiet than the original, hence the peak-normalized signal gives better headroom for later adjustments via the volume property of AudioSource
Load In Background	If enabled, the audio clip will be loading in the background without causing stalls on the main thread. This is off by default in order to ensure the standard Unity behavior where all AudioClips have finished loading when the **scene** starts playing. Note that play requests on AudioClips that are still loading in the background will be deferred until the clip is done loading. The load state can be queried via the AudioClip.loadState property.
Preload Audio Data	If enabled, the audio clip will be pre-loaded when the scene is loaded. This is on by default to reflect standard Unity behavior where all AudioClips have finished loading when the scene starts playing. If this flag is not set, the audio data will either be loaded on the first AudioSource.Play()/AudioSource.PlayOneShot(), or it can be loaded via AudioSource.LoadAudioData() and unloaded again via AudioSource.UnloadAudioData().
Quality	Determines the amount of Compression to be applied to a **Compressed** clip. Does not apply to PCM/ADPCM/HEVAG formats. Statistics about the file size can be seen in the inspector. A good approach to tuning this value is to drag the slider to a place that leaves the playback “good enough” while keeping the file small enough for your distribution requirements. Note that the original size relates to the original file, so if this was an MP3 file and Compression Format is set to PCM (i.e. uncompressed), the resulting Ratio will be bigger than 100% because the file is now stored uncompressed and taking up more space than the source MP3 that it came from.

Preview Window

The Preview window contains three icons.

When Auto Play is on the clips will play as soon as they are selected.

When Loop is on the clips will play in a continual loop.

This will play the clip.

Importing Audio Assets

Unity is able to read a wide range of source file formats. Whenever a file is imported, it will be transcoded to a format suitable for the build target and the type of sound. This is selectable via the Compression Format setting in the inspector.

In general, the PCM and Vorbis/MP3 formats are preferrable for keeping the sound as close to the original as possible. PCM is very lightweight on the CPU requirements, since the sound is uncompressed and can just be read from memory. Vorbis/MP3 allows adaptively discarding less audible information via the Quality slider.

ADPCM is a compromise between memory and CPU usage in that it uses only slightly more CPU than the uncompressed PCM option, but yields a constant 3.5 compression factor, which is in general about 3 times worse than the compression that can be achieved with Vorbis or MP3 compression. Furthermore ADPCM (like PCM) allows automatically optimized or manually set sample rates to be used, which – depending on the frequency content of the sound and the acceptable loss of quality – can further shrink the size of the packed sound assets.

Module files (.mod,.it,.s3m..xm) can deliver very high quality with an extremely low footprint. When using module files, unless you specifically want this, make sure that the Load Type is set to *Compressed In Memory*, because if it's set to *Decompress On Load* the whole song will be decompressed. This is a new behavior in Unity 5.0 that allows GetData/SetData to be used on these types of clips too, but the general and default use case for tracker modules is to have them compressed in memory.

As a general rule of thumb, *Compressed* audio (or modules) are best for long files like background music or dialog, while *PCM* and *ADPCM* is better for short sound effects that contain some noise, as the artefacts of *ADPCM* are too apparent on smooth signals. You should tweak the amount of Compression using the compression slider. Start with high compression and gradually reduce the setting to the point where the loss of sound quality is perceptible. Then, increase it again slightly until the perceived loss of quality disappears.

Platform specific details

Unity supports importing a variety of source format sound files. However when importing these files (with the exception of tracker files), they are always re-encoded to the build target format. By default, this format is Vorbis, though this can be overridden per platform to other formats (ADPCM, MP3 etc) if required.

2017–08–09 Page published with no editorial review

Ambisonic support checkbox added in Unity 2017.1

Audio Listener

Leave feedback

SWITCH TO SCRIPTING

The **Audio Listener** acts as a microphone-like device. It receives input from any given Audio Source in the **scene** and plays sounds through the computer speakers. For most applications it makes the most sense to attach the listener to the Main Camera. If an audio listener is within the boundaries of a Reverb Zone reverberation is applied to all audible sounds in the scene. Furthermore, Audio Effects can be applied to the listener and it will be applied to all audible sounds in the scene.

Properties

The Audio Listener has no properties. It simply must be added to work. It is always added to the Main Camera by default.

Details

The Audio Listener works in conjunction with Audio Sources, allowing you to create the aural experience for your games. When the Audio Listener is attached to a **GameObject** in your scene, any Sources that are close enough to the Listener will be picked up and output to the computer's speakers. Each scene can only have 1 Audio Listener to work properly.

If the Sources are 3D (see import settings in Audio Clip), the Listener will emulate position, velocity and orientation of the sound in the 3D world (You can tweak attenuation and 3D/2D behavior in great detail in Audio Source) . 2D will ignore any 3D processing. For example, if your character walks off a street into a night club, the night club's music should probably be 2D, while the individual voices of characters in the club should be mono with their realistic positioning being handled by Unity.

You should attach the Audio Listener to either the Main Camera or to the GameObject that represents the player. Try both to find what suits your game best.

Hints

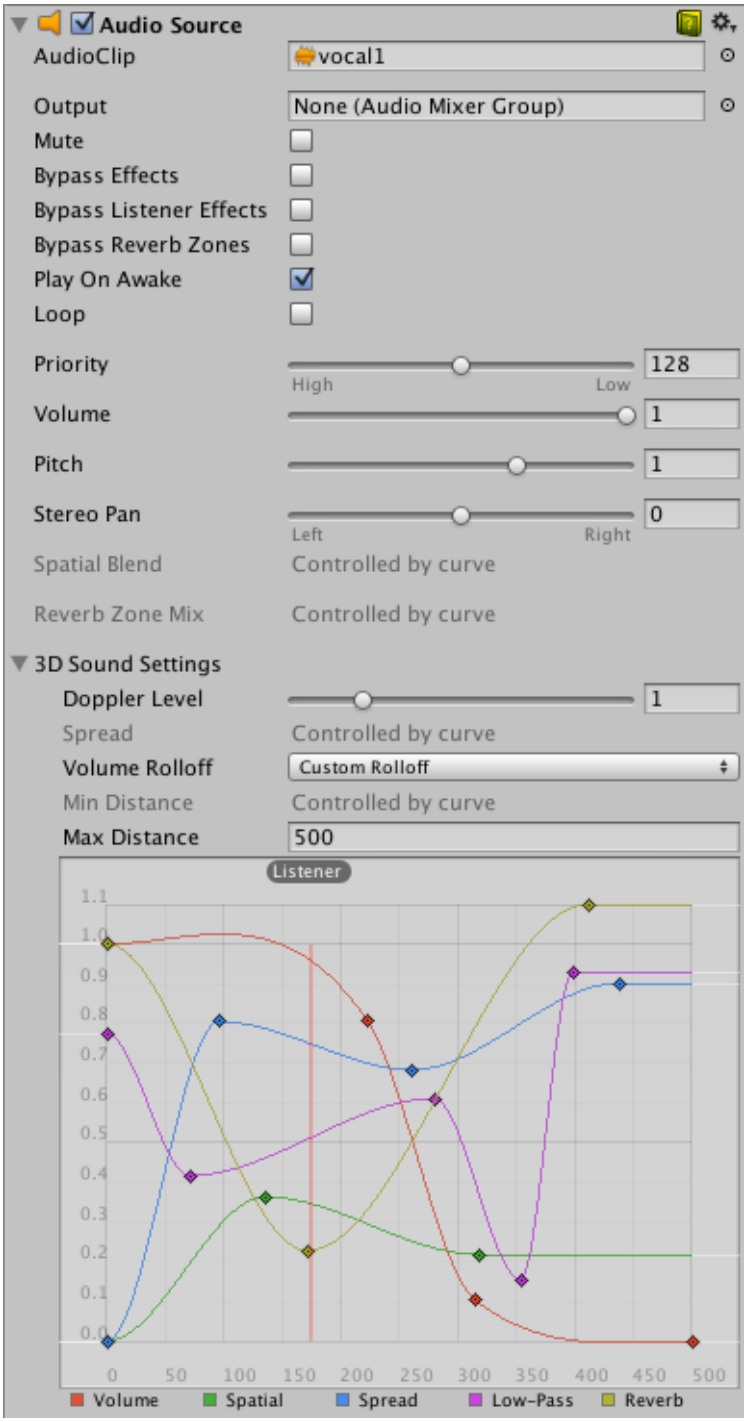
Each scene can only have one Audio Listener.
You access the project-wide audio settings using the Audio Manager, found in the **Edit->Project Settings->Audio** menu.
View the Audio Clip **Component** page for more information about Mono vs Stereo sounds.

Audio Source

Leave feedback

SWITCH TO SCRIPTING

The **Audio Source** plays back an Audio Clip in the **scene**. The clip can be played to an audio listener or through an audio mixer. The audio source can play any type of Audio Clip and can be configured to play these as 2D, 3D, or as a mixture (*SpatialBlend*). The audio can be spread out between speakers (stereo to 7.1) (*Spread*) and morphed between 3D and 2D (*SpatialBlend*). This can be controlled over distance with falloff curves. Also, if the listener is within one or multiple Reverb Zones, reverberation is applied to the source. Individual filters can be applied to each audio source for an even richer audio experience. See Audio Effects for more details.



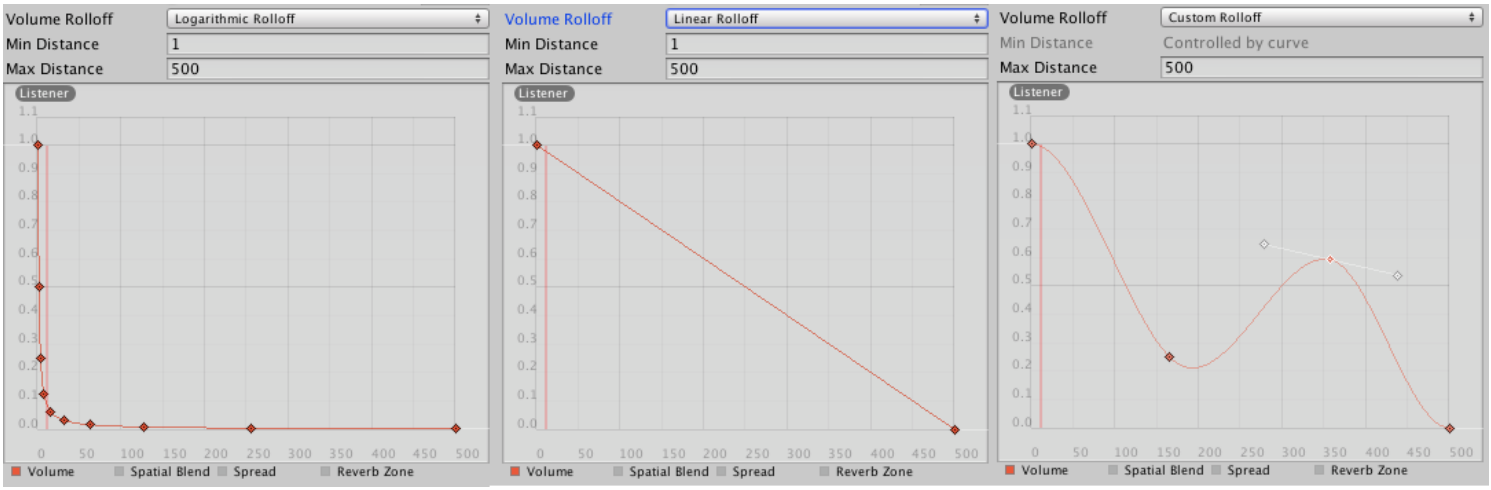
Properties

Property:	*Function:*
Audio Clip	Reference to the sound clip file that will be played.
Output	The sound can be output through an audio listener or an audio mixer.
Mute	If enabled the sound will be playing but muted.
Bypass Effects	This is to quickly "by-pass" filter effects applied to the audio source. An easy way to turn all effects on/off.
Bypass Listener Effects	This is to quickly turn all Listener effects on/off.
Bypass Reverb Zones	This is to quickly turn all Reverb Zones on/off.
Play On Awake	If enabled, the sound will start playing the moment the scene launches. If disabled, you need to start it using the **Play()** command from scripting.
Loop	Enable this to make the **Audio Clip** loop when it reaches the end.
Priority	Determines the priority of this audio source among all the ones that coexist in the scene. (Priority: 0 = most important. 256 = least important. Default = 128.). Use 0 for music tracks to avoid it getting occasionally swapped out.
Volume	How loud the sound is at a distance of one world unit (one meter) from the **Audio Listener**.
Pitch	Amount of change in pitch due to slowdown/speed up of the **Audio Clip**. Value 1 is normal playback speed.
Stereo Pan	Sets the position in the stereo field of 2D sounds.
Spatial Blend	Sets how much the 3D engine has an effect on the audio source.
Reverb Zone Mix	Sets the amount of the output signal that gets routed to the reverb zones. The amount is linear in the (0 - 1) range, but allows for a 10 dB amplification in the (1 - 1.1) range which can be useful to achieve the effect of near-field and distant sounds.
3D Sound Settings	Settings that are applied proportionally to the Spatial Blend parameter.
Doppler Level	Determines how much doppler effect will be applied to this audio source (if is set to 0, then no effect is applied).
Spread	Sets the spread angle to 3D stereo or multichannel sound in speaker space.
Min Distance	Within the MinDistance, the sound will stay at loudest possible. Outside MinDistance it will begin to attenuate. Increase the MinDistance of a sound to make it 'louder' in a 3d world, and decrease it to make it 'quieter' in a 3d world.
Max Distance	The distance where the sound stops attenuating at. Beyond this point it will stay at the volume it would be at MaxDistance units from the listener and will not attenuate any more.
Rolloff Mode	How fast the sound fades. The higher the value, the closer the Listener has to be before hearing the sound. (This is determined by a Graph).
- Logarithmic Rolloff	The sound is loud when you are close to the audio source, but when you get away from the object it decreases significantly fast.
- Linear Rolloff	The further away from the audio source you go, the less you can hear it.

Property:	*Function:*
- Custom Rolloff	The sound from the audio source behaves accordingly to how you set the graph of roll offs.

Types of Rolloff

There are three Rolloff modes: Logarithmic, Linear and Custom Rolloff. The Custom Rolloff can be modified by modifying the volume distance curve as described below. If you try to modify the volume distance function when it is set to Logarithmic or Linear, the type will automatically change to Custom Rolloff.



Rolloff Modes that an audio source can have.

Distance Functions

There are several properties of the audio that can be modified as a function of the distance between the audio source and the audio listener.

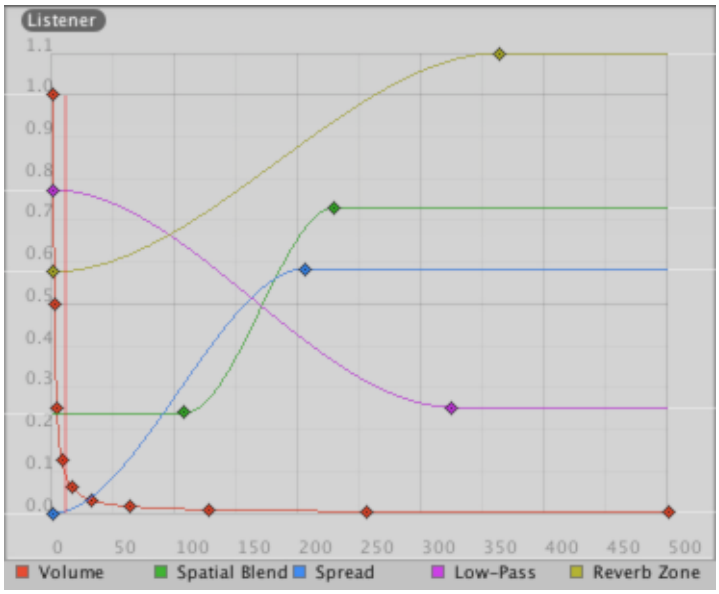
Volume: Amplitude(0.0 - 1.0) over distance.

Spatial Blend: 2D (original channel mapping) to 3D (all channels downmixed to mono and attenuated according to distance and direction).

Spread: Angle (degrees 0.0 - 360.0) over distance.

Low-Pass (only if LowPassFilter is attached to the AudioSource): Cutoff Frequency (22000.0–10.0) over distance.

Reverb Zone: Amount of signal routed to the reverb zones. Note that the volume property and distance and directional attenuation are applied to the signal first and therefore affect both the direct and reverberated signals.



Distance functions for Volume, Spatial Blend, Spread, Low-Pass audio filter, and Reverb Zone Mix. The current distance to the Audio Listener is marked in the graph by the red vertical line.

To modify the distance functions, you can edit the curves directly. For more information, see the guide to Editing Curves.

Creating Audio Sources

Audio Sources don't do anything without an assigned **Audio Clip**. The Clip is the actual sound file that will be played back. The Source is like a controller for starting and stopping playback of that clip, and modifying other audio properties.

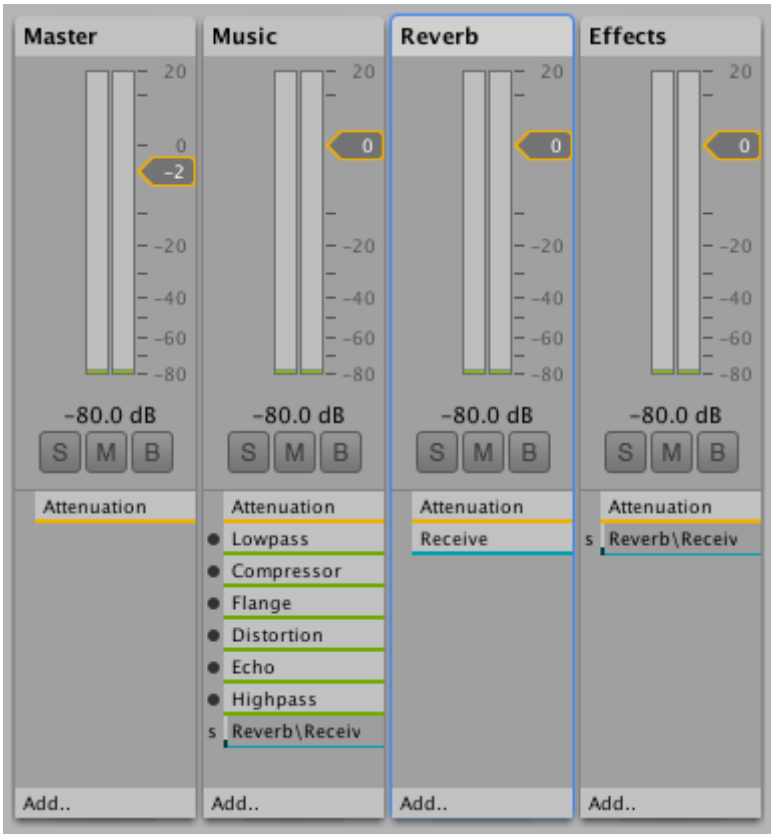
To create a new Audio Source:

Import your audio files into your Unity Project. These are now Audio Clips.
Go to **GameObject->Create Empty** from the menubar.
With the new GameObject selected, select **Component->Audio->Audio Source**.
Assign the **Audio Clip** property of the Audio Source **Component** in the **Inspector**.

Note: If you want to create an **Audio Source** just for one **Audio Clip** that you have in the **Assets** folder then you can just drag that clip to the **scene view** - a **GameObject** with an **Audio Source** component will be created automatically for it. Dragging a clip onto on existing **GameObject** will attach the clip along with a new **Audio Source** if there isn't one already there. If the object does already have an **Audio Source** then the newly dragged clip will replace the one that the source currently uses.

Audio Mixer

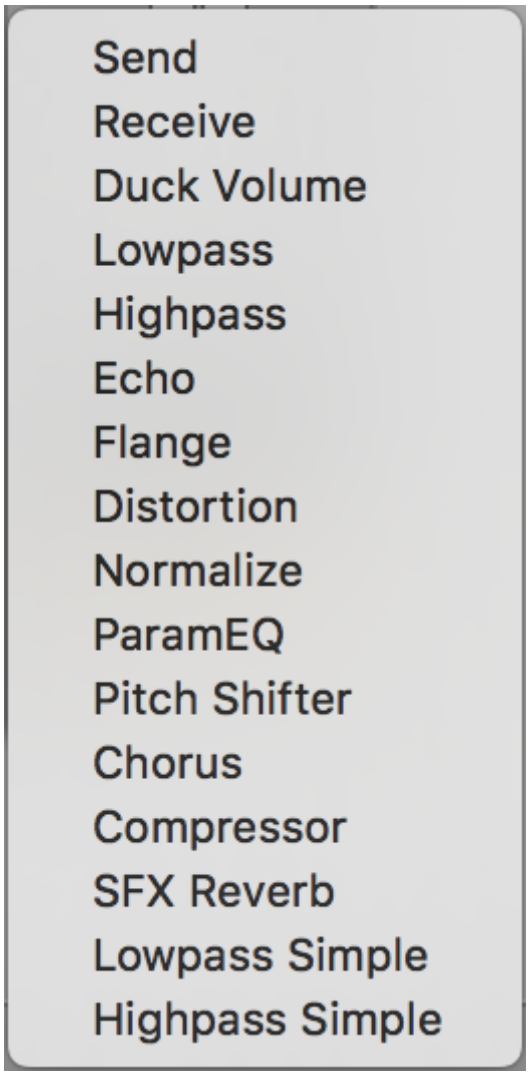
Leave feedback



Properties

Property:	*Function:*
S Button	Soloing the group.
M Button	Muting the group.
B Button	Bypassing all the effects in the group.

When you select **Add** you can select from a list of effects that can be applied to the group.

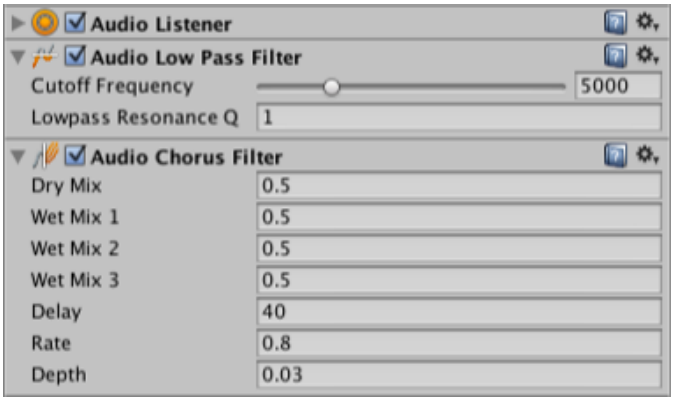

Send
Receive
Duck Volume
Lowpass
Highpass
Echo
Flange
Distortion
Normalize
ParamEQ
Pitch Shifter
Chorus
Compressor
SFX Reverb
Lowpass Simple
Highpass Simple

Audio Filters

Leave feedback

You can modify the output of Audio Source and Audio Listener **components** by applying **Audio Effects**. These can filter the frequency ranges of the sound or apply reverb and other effects.

The effects are applied by adding effect components to the object with the Audio Source or Audio Listener. The ordering of the components is important, since it reflects the order in which the effects will be applied to the source audio. For example, in the image below, an Audio Listener is modified first by an **Audio Low Pass Filter** and then an Audio Chorus Filter.



To change the ordering of these and any other components, open a context menu in the **inspector** and select the *Move Up* or *Move Down* commands. Enabling or disabling an effect component determines whether it will be applied or not.

Though highly optimized, some filters are still CPU intensive. Audio CPU usage can monitored in the profiler under the Audio Tab.

See the other pages in this section for further information about the specific filter types available.

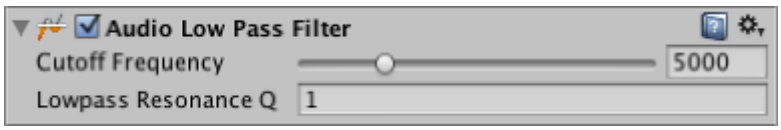
Audio Low Pass Filter

Leave feedback

SWITCH TO SCRIPTING

The **Audio Low Pass Filter** passes low frequencies of an AudioSource or all sound reaching an AudioListener while removing frequencies higher than the **Cutoff Frequency**.

Properties



Property:	*Function:*
Cutoff Frequency	Lowpass cutoff frequency in Hertz (range 0.0 to 22000.0, default = 5000.0).
Lowpass Resonance Q	Lowpass resonance quality value (range 1.0 to 10.0, default = 1.0).

Details

The **Lowpass Resonance Q** (short for Lowpass Resonance Quality Factor) determines how much the filter's self-resonance is dampened. Higher lowpass resonance quality indicates a lower rate of energy loss, that is the oscillations die out more slowly.

The **Audio Low Pass Filter** has a Rolloff curve associated with it, making it possible to set **Cutoff Frequency** over distance between the AudioSource and the AudioListener.

Sounds propagates very differently given the environment. For example, to compliment a visual fog effect add a subtle low-pass to the **Audio Listener**. The high frequencies of a sound being emitted from behind a door will be filtered out by the door and so won't reach the listener. To simulate this, simply change the Cutoff Frequency when opening the door.

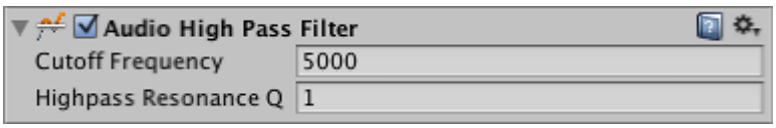
Audio High Pass Filter

Leave feedback

SWITCH TO SCRIPTING

The **Audio High Pass Filter** passes high frequencies of an AudioSource and cuts off signals with frequencies lower than the **Cutoff Frequency**.

Properties



Property:	*Function:*
Cutoff Frequency	Highpass cutoff frequency in Hertz (range 10.0 to 22000.0, default = 5000.0).
Highpass Resonance Q	Highpass resonance quality value (range 1.0 to 10.0, default = 1.0).

Details

The **Highpass resonance Q** (short for Highpass Resonance Quality Factor) determines how much the filter's self-resonance is dampened. Higher highpass resonance quality indicates a lower rate of energy loss, that is the oscillations die out more slowly.

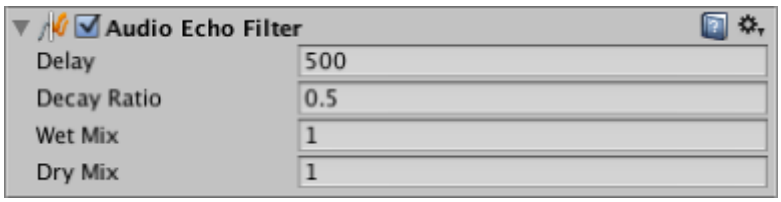
Audio Echo Filter

Leave feedback

SWITCH TO SCRIPTING

The **Audio Echo Filter** repeats a sound after a given **Delay**, attenuating the repetitions based on the **Decay Ratio**.

Properties



Property:	*Function:*
Delay	Echo delay in ms. 10 to 5000. Default = 500.
Decay Ratio	Echo decay per delay. 0 to 1. 1.0 = No decay, 0.0 = total decay (ie simple 1 line delay). Default = 0.5.L
Wet Mix	Volume of echo signal to pass to output. 0.0 to 1.0. Default = 1.0.
Dry Mix	Volume of original signal to pass to output. 0.0 to 1.0. Default = 1.0.

Details

The **Wet Mix** value determines the amplitude of the filtered signal, where the **Dry Mix** determines the amplitude of the unfiltered sound output.

Hard surfaces reflects the propagation of sound. For example a large canyon can be made more convincing with the Audio Echo Filter.

Sound propagates slower than light - we all know that from lightning and thunder. To simulate this, add an Audio Echo Filter to an event sound, set the Wet Mix to 0.0 and modulate the Delay to the distance between AudioSource and AudioListener.

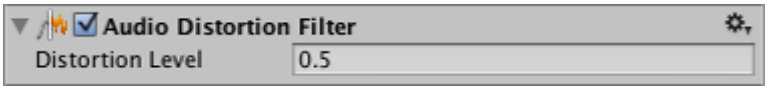
Audio Distortion Filter

Leave feedback

SWITCH TO SCRIPTING

The **Audio Distortion Filter** distorts the sound from an AudioSource or sounds reaching the AudioListener.

Properties



Property:	*Function:*
Distortion	Distortion value. 0.0 to 1.0. Default = 0.5.

Details

Apply the **Audio Distortion Filter** to simulate the sound of a low quality radio transmission.

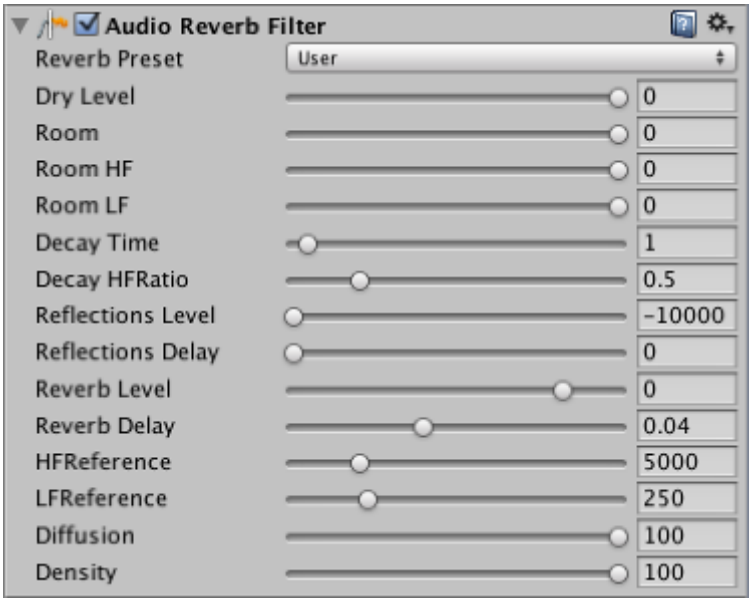
Audio Reverb Filter

Leave feedback

SWITCH TO SCRIPTING

The **Audio Reverb Filter** takes an Audio Clip and distorts it to create a custom reverb effect.

Properties



Property:	*Function:*
Reverb Preset	Custom reverb presets, select user to create your own customized reverbs.
Dry Level	Mix level of dry signal in output in mB. Ranges from –10000.0 to 0.0. Default is 0.
Room	Room effect level at low frequencies in mB. Ranges from –10000.0 to 0.0. Default is 0.0.
Room HF	Room effect high-frequency level in mB. Ranges from –10000.0 to 0.0. Default is 0.0.
Room LF	Room effect low-frequency level in mB. Ranges from –10000.0 to 0.0. Default is 0.0.
Decay Time	Reverberation decay time at low-frequencies in seconds. Ranges from 0.1 to 20.0. Default is 1.0.
Decay HFRatio	Decay HF Ratio : High-frequency to low-frequency decay time ratio. Ranges from 0.1 to 2.0. Default is 0.5.
Reflections Level	Early reflections level relative to room effect in mB. Ranges from –10000.0 to 1000.0. Default is –10000.0.
Reflections Delay	Early reflections delay time relative to room effect in mB. Ranges from 0 to 0.3. Default is 0.0.
Reverb Level	Late reverberation level relative to room effect in mB. Ranges from –10000.0 to 2000.0. Default is 0.0.
Reverb Delay	Late reverberation delay time relative to first reflection in seconds. Ranges from 0.0 to 0.1. Default is 0.04.
HFReference	Reference high frequency in Hz. Ranges from 1000.0 to 20000.0. Default is 5000.0.
LFReference	Reference low-frequency in Hz. Ranges from 20.0 to 1000.0. Default is 250.0.

Property:	*Function:*
Diffusion	Reverberation diffusion (echo density) in percent. Ranges from 0.0 to 100.0. Default is 100.0.
Density	Reverberation density (modal density) in percent. Ranges from 0.0 to 100.0. Default is 100.0.

Note: These values can only be modified if your **Reverb Preset** is set to **User**, else these values will be grayed out and they will have default values for each preset.

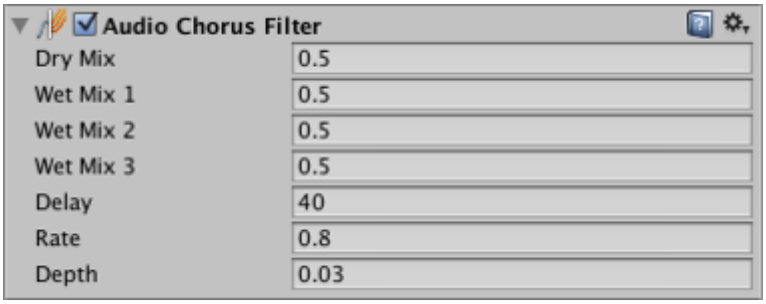
Audio Chorus Filter

Leave feedback

SWITCH TO SCRIPTING

The **Audio Chorus Filter** takes an Audio Clip and processes it creating a chorus effect.

Properties



Property:	*Function:*
Dry Mix	Volume of original signal to pass to output. 0.0 to 1.0. Default = 0.5.
Wet Mix 1	Volume of 1st chorus tap. 0.0 to 1.0. Default = 0.5.
Wet Mix 2	Volume of 2nd chorus tap. This tap is 90 degrees out of phase of the first tap. 0.0 to 1.0. Default = 0.5.
Wet Mix 3	Volume of 3rd chorus tap. This tap is 90 degrees out of phase of the second tap. 0.0 to 1.0. Default = 0.5.
Delay	The LFO's delay in ms. 0.1 to 100.0. Default = 40.0 ms
Rate	The LFO's modulation rate in Hz. 0.0 to 20.0. Default = 0.8 Hz.
Depth	Chorus modulation depth. 0.0 to 1.0. Default = 0.03.
Feed Back	Chorus feedback. Controls how much of the wet signal gets fed back into the filter's buffer. 0.0 to 1.0. Default = 0.0.

Details

The chorus effect modulates the original sound by a sinusoid low frequency oscillator (LFO). The output sounds like there are multiple sources emitting the same sound with slight variations - resembling a choir.

You can tweak the chorus filter to create a flanger effect by lowering the feedback and decreasing the delay, as the flanger is a variant of the chorus.

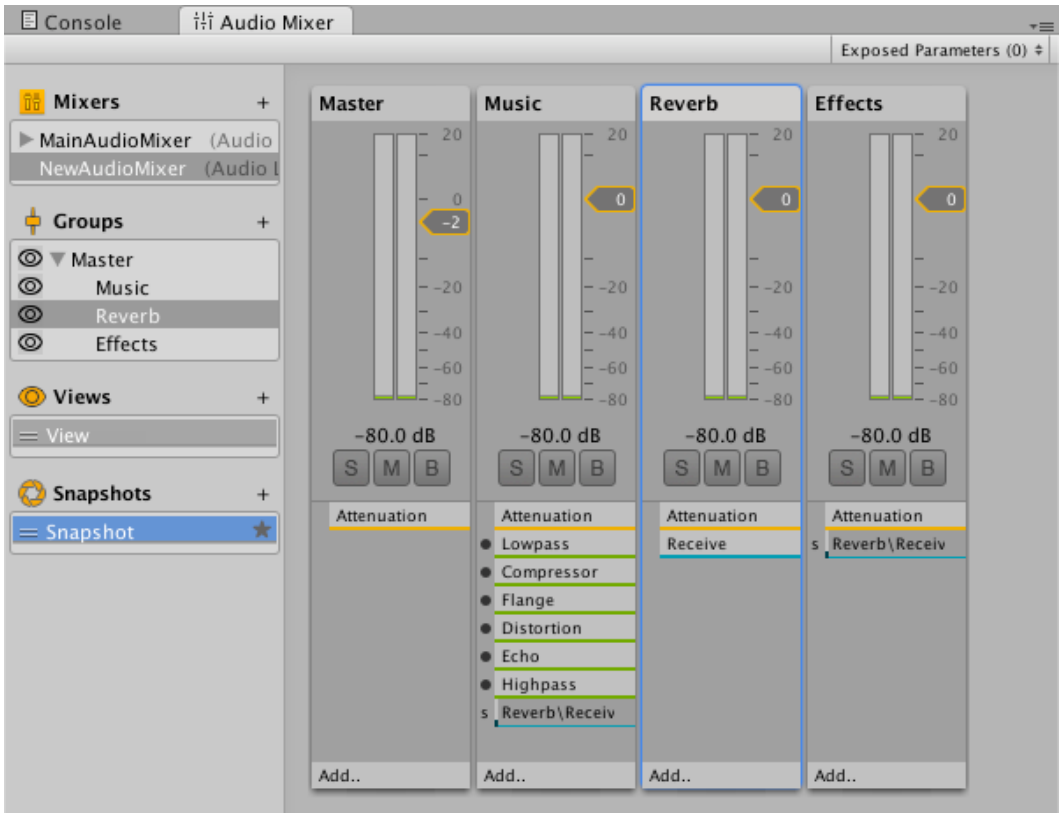
Creating a simple, dry echo is done by setting **Rate** and **Depth** to 0 and tweaking the mixes and **Delay**

Audio Effects

Leave feedback

You can modify the output of Audio Mixer **components** by applying **Audio Effects**. These can filter the frequency ranges of the sound or apply reverb and other effects.

The effects are applied by adding effect components to the relevant section of the Audio Mixer. The ordering of the components is important, since it reflects the order in which the effects will be applied to the source audio. For example, in the image below, the Music section of an Audio Mixer is modified first by a Lowpass effect and then a compressor Effect, Flange Effect and so on.



To change the ordering of these and any other components, open a context menu in the **inspector** and select the *Move Up* or *Move Down* commands. Enabling or disabling an effect component determines whether it will be applied or not.

Though highly optimized, some filters are still CPU intensive. Audio CPU usage can monitored in the profiler under the Audio Tab.

See the other pages in this section for further information about the specific effect types available.

Audio Low Pass Effect

Leave feedback

The **Audio Low Pass Effect** passes low frequencies of an AudioMixer group while removing frequencies higher than the **Cutoff Frequency**.

Properties

Property:	*Function:*
Cutoff freq	Lowpass cutoff frequency in Hertz (range 10.0 to 22000.0, default = 5000.0).
Resonance	Lowpass resonance quality value (range 1.0 to 10.0, default = 1.0).

Details

The **Resonance** (short for Lowpass Resonance Quality Factor) determines how much the filter's self-resonance is dampened. Higher lowpass resonance quality indicates a lower rate of energy loss, that is the oscillations die out more slowly.

Audio High Pass Effect

Leave feedback

The **Highpass Effect** passes high frequencies of an AudioMixer group and cuts off signals with frequencies lower than the **Cutoff Frequency**.

Properties

Property:	*Function:*
Cutoff freq	Highpass cutoff frequency in Hertz (range 10.0 to 22000.0, default = 5000.0).
Resonance	Highpass resonance quality value (range 1.0 to 10.0, default = 1.0).

Details

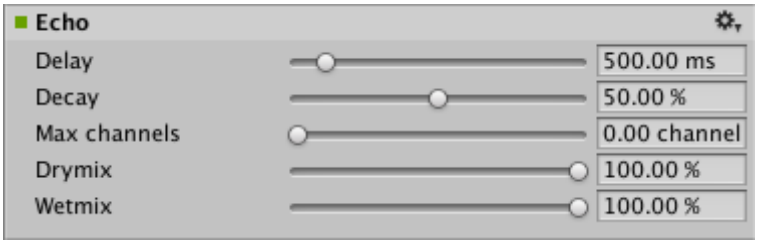
The **Resonance** (short for Highpass Resonance Quality Factor) determines how much the filter's self-resonance is dampened. Higher highpass resonance quality indicates a lower rate of energy loss, that is the oscillations die out more slowly.

Audio Echo Effect

Leave feedback

The **Audio Echo Effect** repeats a sound after a given **Delay**, attenuating the repetitions based on the **Decay Ratio**.

Properties



Property:	*Function:*
Delay	Echo delay in ms. 10 to 5000. Default = 500.
Decay	Echo decay per delay. 0 to 100%. 100% = No decay, 0% = total decay (ie simple 1 line delay). Default = 50%.
Max channels	
Drymix	Volume of original signal to pass to output. 0 to 100%. Default = 100%.
Wetmix	Volume of echo signal to pass to output. 0 to 100%. Default = 100%.

Details

The **Wetmix** value determines the amplitude of the filtered signal, where the **Drymix** determines the amplitude of the unfiltered sound output.

Hard surfaces reflects the propagation of sound. For example a large canyon can be made more convincing with the Audio Echo Filter.

Audio Flange Effect

Leave feedback

The **Audio Flange Effect** is used to create the **audio effect** produced by mixing two identical signals together, one signal delayed by a small and gradually changing period, usually smaller than 20 milliseconds.

Properties

Property: Function:

Drymix Percentage of original signal to pass to output. 0.0 to 100.0%. Default = 45%.

Wetmix Percentage of flange signal to pass to output. 0.0 to 100.0%. Default = 55%.

Depth 0.01 to 1.0. Default = 1.0.

Rate 0.1 to 20 Hz. Default = 10 Hz.

Audio Distortion Effect

Leave feedback

The **Distortion Effect** distorts the sound from an AudioMixer group.

Properties

Property:	*Function:*
Distortion	Distortion value. 0.0 to 1.0. Default = 0.5.

Details

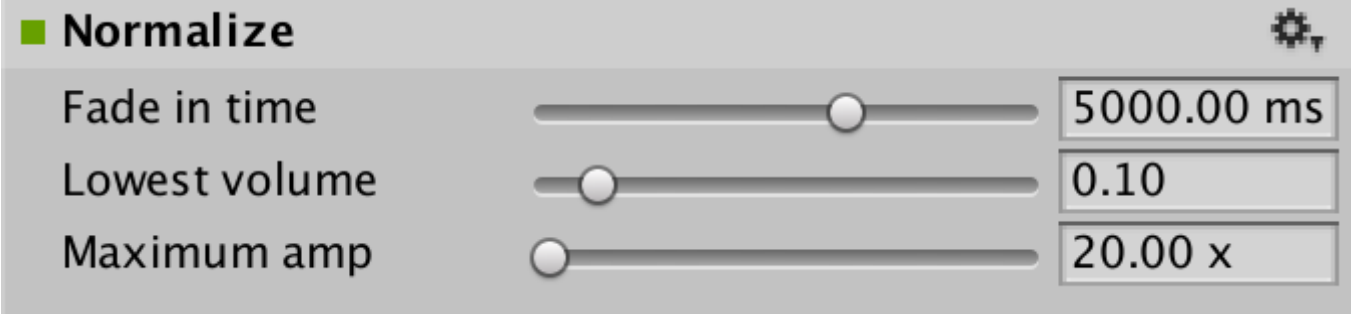
Apply the **Distortion Effect** to simulate the sound of a low quality radio transmission.

Audio Normalize Effect

Leave feedback

The **Audio Normalize Effect** applies a constant amount of gain to an audio stream to bring the average or peak amplitude to a target level.

Properties



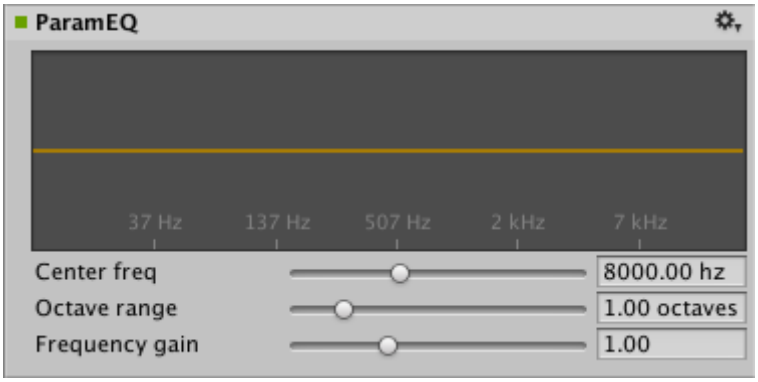
Property:	*Function:*
Fade in time	Fade in time of the effect in milliseconds (range 0 to 20000.0, default = 5000.0 milliseconds).
Lowest volume	(range 0.0 to 1.0, default = 0.10).
Maximum amp	Maximum amplification (range 0.0 to 100000.0, default = 20 x).

Audio Parametric Equalizer Effect

Leave feedback

The **Audio Param EQ Effect** is used to alter the frequency response of an audio system using linear filters.

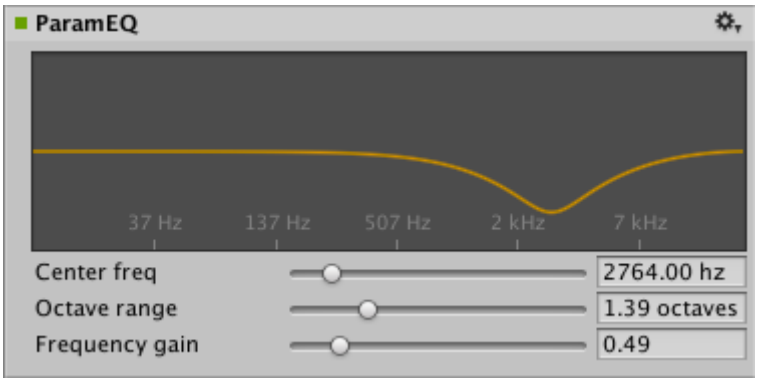
Properties



Property:	*Function:*
Center freq	The frequency in Hertz where the gain is applied (range 20.0 to 22000.0, default = 8000.0 Hz).
Octave Range	The number of Octaves over which the gain is applied (Centered on the Center Frequency) (range 0.20 to 5.00, default = 1.0 octave).
Frequency Gain	The gain applied (range 0.05 to 3.00, default = 1.00 - no gain applied).

Details

The graph shows the effect of the gain to be applied over the frequency range of the audio output.



Audio Pitch Shifter Effect

Leave feedback

The **Audio Pitch Shifter Effect** is used to shift a signal up or down in pitch.

Properties

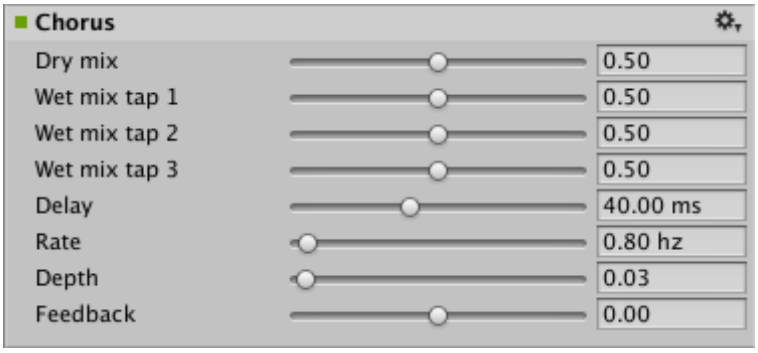
Property:	*Function:*
Pitch	The pitch multiplier (range 0.5 x to 2.0 x, default 1.0 x).
FFT Size	(range 256.0 to 4096.0, default = 1024.0).
Overlap	(range 1 to 32, default = 4).
Max channels	The maximum number of channels (range 0 to 16, default = 0 channels).

Audio Chorus Effect

Leave feedback

The **Audio Chorus Effect** takes an Audio Mixer group output and processes it creating a chorus effect.

Properties



Property:	*Function:*
Dry mix	Volume of original signal to pass to output. 0.0 to 1.0. Default = 0.5.
Wet mix tap 1	Volume of 1st chorus tap. 0.0 to 1.0. Default = 0.5.
Wet mix tap 2	Volume of 2nd chorus tap. This tap is 90 degrees out of phase of the first tap. 0.0 to 1.0. Default = 0.5.
Wet mix tap 3	Volume of 3rd chorus tap. This tap is 90 degrees out of phase of the second tap. 0.0 to 1.0. Default = 0.5.
Delay	The LFO's delay in ms. 0.1 to 100.0. Default = 40.0 ms
Rate	The LFO's modulation rate in Hz. 0.0 to 20.0. Default = 0.8 Hz.
Depth	Chorus modulation depth. 0.0 to 1.0. Default = 0.03.
Feedback	Chorus feedback. Controls how much of the wet signal gets fed back into the filter's buffer. 0.0 to 1.0. Default = 0.0.

Details

The chorus effect modulates the original sound by a sinusoid low frequency oscillator (LFO). The output sounds like there are multiple sources emitting the same sound with slight variations - resembling a choir.

You can tweak the chorus filter to create a flanger effect by lowering the feedback and decreasing the delay, as the flanger is a variant of the chorus.

Creating a simple, dry echo is done by setting **Rate** and **Depth** to 0 and tweaking the mixes and **Delay**

Audio Compressor Effect

Leave feedback

The **Audio Compressor Effect** reduces the volume of loud sounds or amplifies quiet sounds by narrowing or "compressing" an audio signal's dynamic range.

Properties

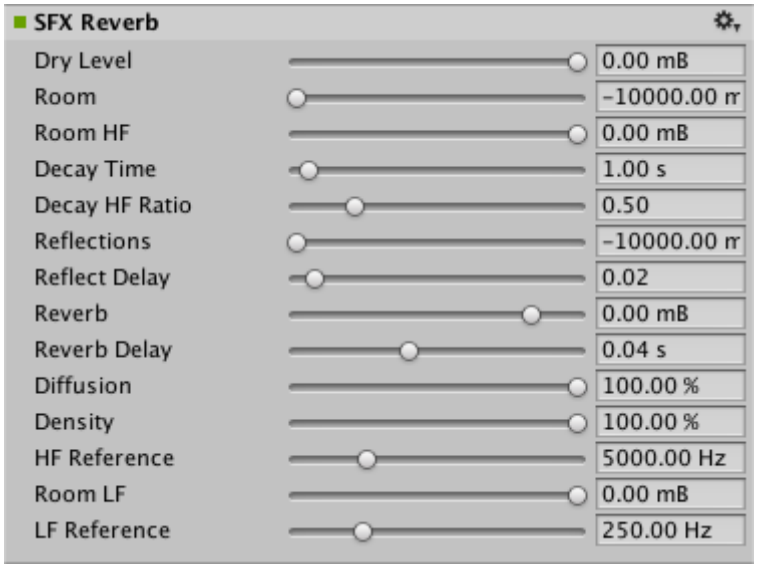
Property:	*Function:*
Threshold	Threshold level in dB (range 0 to –60dB, default = 0dB).
Attack	The rate the effect is applied in ms. (range 10.0 to 200.0 ms, default = 50.0 ms).
Release	The rate the effect is released in ms. (range 20.0 to 1000.0 ms, default = 50.0 ms).
Make up gain	Make up gain level in dB (range 0 to 30dB, default = 0dB).

Audio SFX Reverb Effect

Leave feedback

The **SFX Reverb Effect** takes the output of an Audio Mixer group and distorts it to create a custom reverb effect.

Properties



Property:	*Function:*
Dry Level	Mix level of dry signal in output in mB. Ranges from –10000.0 to 0.0. Default is 0 mB.
Room	Room effect level at low frequencies in mB. Ranges from –10000.0 to 0.0. Default is –10000.0 mB.
Room HF	Room effect high-frequency level in mB. Ranges from –10000.0 to 0.0. Default is 0.0 mB.
Decay Time	Reverberation decay time at low-frequencies in seconds. Ranges from 0.1 to 20.0. Default is 1.0.
Decay HF Ratio	Decay HF Ratio : High-frequency to low-frequency decay time ratio. Ranges from 0.1 to 2.0. Default is 0.5.
Reflections	Early reflections level relative to room effect in mB. Ranges from –10000.0 to 1000.0. Default is –10000.0 mB.
Reflect Delay	Early reflections delay time relative to room effect in mB. Ranges from –10000.0 to 2000.0. Default is 0.02.
Reverb	Late reverberation level relative to room effect in mB. Ranges from –10000.0 to 2000.0. Default is 0.0 mB.
Reverb Delay	Late reverberation delay time relative to first reflection in seconds. Ranges from 0.0 to 0.1. Default is 0.04 s.
Diffusion	Reverberation diffusion (echo density) in percent. Ranges from 0.0 to 100.0. Default is 100.0%.
Density	Reverberation density (modal density) in percent. Ranges from 0.0 to 100.0. Default is 100.0%.
HFReference	Reference high frequency in Hz. Ranges from 20.0 to 20000.0. Default is 5000.0 Hz.
Room LF	Room effect low-frequency level in mB. Ranges from –10000.0 to 0.0. Default is 0.0 mB.

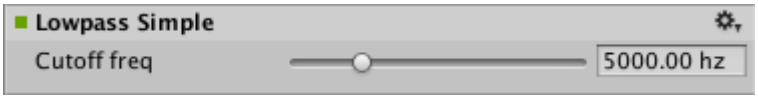
Property:	*Function:*
LFReference	Reference low-frequency in Hz. Ranges from 20.0 to 1000.0. Default is 250.0 Hz.

Audio Low Pass Simple Effect

Leave feedback

The **Audio Low Pass Simple Effect** passes low frequencies of an AudioMixer group while removing frequencies higher than the **Cutoff Frequency**.

Properties



Property:	*Function:*
Cutoff freq	Lowpass cutoff frequency in Hertz (range 10.0 to 22000.0, default = 5000.0).

Details

The **Resonance** (short for Lowpass Resonance Quality Factor) determines how much the filter's self-resonance is dampened. Higher lowpass resonance quality indicates a lower rate of energy loss, that is the oscillations die out more slowly.

For additional control over the resonance value of the low pass filter use the Audio Low Pass effect.

Audio High Pass Simple Effect

Leave feedback

The **Highpass Simple Effect** passes high frequencies of an AudioMixer group and cuts off signals with frequencies lower than the **Cutoff Frequency**.

Properties

Property:	*Function:*
Cutoff freq	Highpass cutoff frequency in Hertz (range 10.0 to 22000.0, default = 5000.0).

Details

The **Resonance** (short for Highpass Resonance Quality Factor) determines how much the filter's self-resonance is dampened. Higher highpass resonance quality indicates a lower rate of energy loss, that is the oscillations die out more slowly.

For additional control over the resonance value of the high pass filter use the Audio High Pass effect.

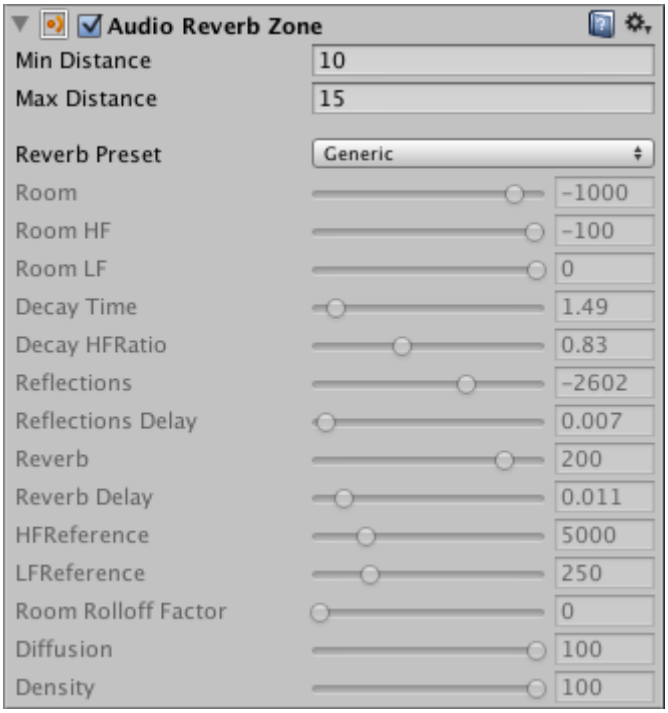
Reverb Zones

Leave feedback

SWITCH TO SCRIPTING

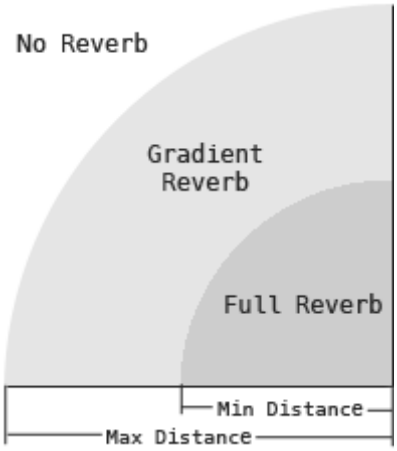
Reverb Zones take an Audio Clip and distorts it depending where the **audio listener** is located inside the reverb zone. They are used when you want to gradually change from a point where there is no ambient effect to a place where there is one, for example when you are entering a cavern.

Properties



Property:	*Function:*
Min Distance	Represents the radius of the inner circle in the **gizmo**, this determines the zone where there is a gradually reverb effect and a full reverb zone.
Max Distance	Represents the radius of the outer circle in the gizmo, this determines the zone where there is no effect and where the reverb starts to get applied gradually.
Reverb Preset	Determines the reverb effect that will be used by the reverb zone.

This diagram illustrates the properties of the reverb zone.



How the sound works in a reverb zone

Hints

You can mix reverb zones to create combined effects.

Microphone

Leave feedback

SWITCH TO SCRIPTING

The **Microphone** class is useful for capturing input from a built-in (*physical*) microphone on your PC or mobile device.

With this class, you can start and end a recording from a built-in microphone, get a listing of available audio input devices (microphones), and find out the status of each such input device.

There is no **component** for the Microphone class but you can access it from a script. See the Microphone page in the Scripting Reference for further information.

Audio Settings

Leave feedback

SWITCH TO SCRIPTING

The AudioSettings class contains various bits of global information relating to the sound system, but most importantly it contains API that allows resetting the audio system at runtime in order to change settings such as speaker mode, sample rate (if supported by the platform), DSP buffer size and real/virtual voice counts.

Many of these settings can also be configured in the Audio section of the **project settings**. When changed these settings will apply both to the editor and define the initial state of the game while changes performed using the AudioSettings API only apply to the runtime of the game and are reset back to the state defined in the **project settings** when stopping the game in the editor.

The game may provide a sound options menu in which the user can change the sound settings or changes may come from outside in response to a device change such as plugging in an external audio input/output device or even an HDMI monitor which may also act as an audio device. The AudioConfiguration AudioSettings.GetConfiguration() / bool AudioSettings.Reset(AudioConfiguration config) API can read and apply global changes to the current sound system configuration and essentially replaces the AudioSettings.SetDSPBufferSize(...) function and the AudioSettings.outputSampleRate, AudioSettings.speakerMode which had the side effect of reinitializing the whole audio system when the properties were modified.

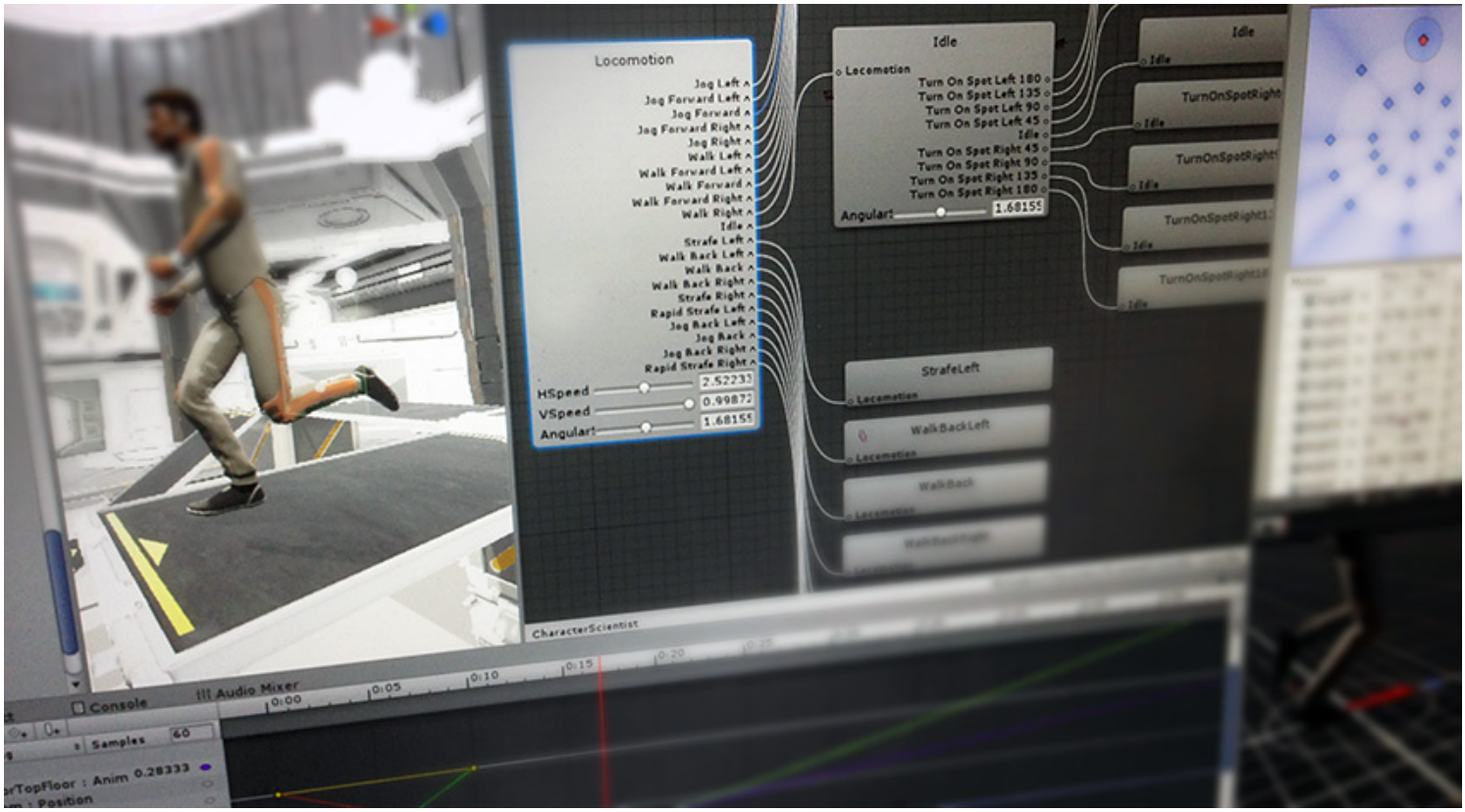
The API defines the AudioSettings.OnAudioConfigurationChanged(bool device) to set up a callback through which **scripts** can be notified about audio configuration changes. These can either be caused by actual device changes or by a configuration initiated by script.

It is important to note that whenever runtime modifications of the global audio system configuration are performed all audio objects have to be reloaded. This works for disk-based AudioClip **assets** and audio mixers, but any AudioClips generated or modified by scripts are lost and have to be recreated. Likewise any play state is lost too, so these need to be regenerated in the AudioSettings.OnAudioConfigurationChanged(...) callback.

For details and examples see the scripting API reference.

Animation

Leave feedback



Animation in Unity

Unity's Animation features include retargetable animations, full control of animation weights at runtime, event calling from within the animation playback, sophisticated **state machine** hierarchies and transitions, blend shapes for facial animations, and much more.

Read this section to find out how to import and work with imported animation, and how to animate objects, colours, and any other parameters within Unity itself.

Related tutorials: Animation

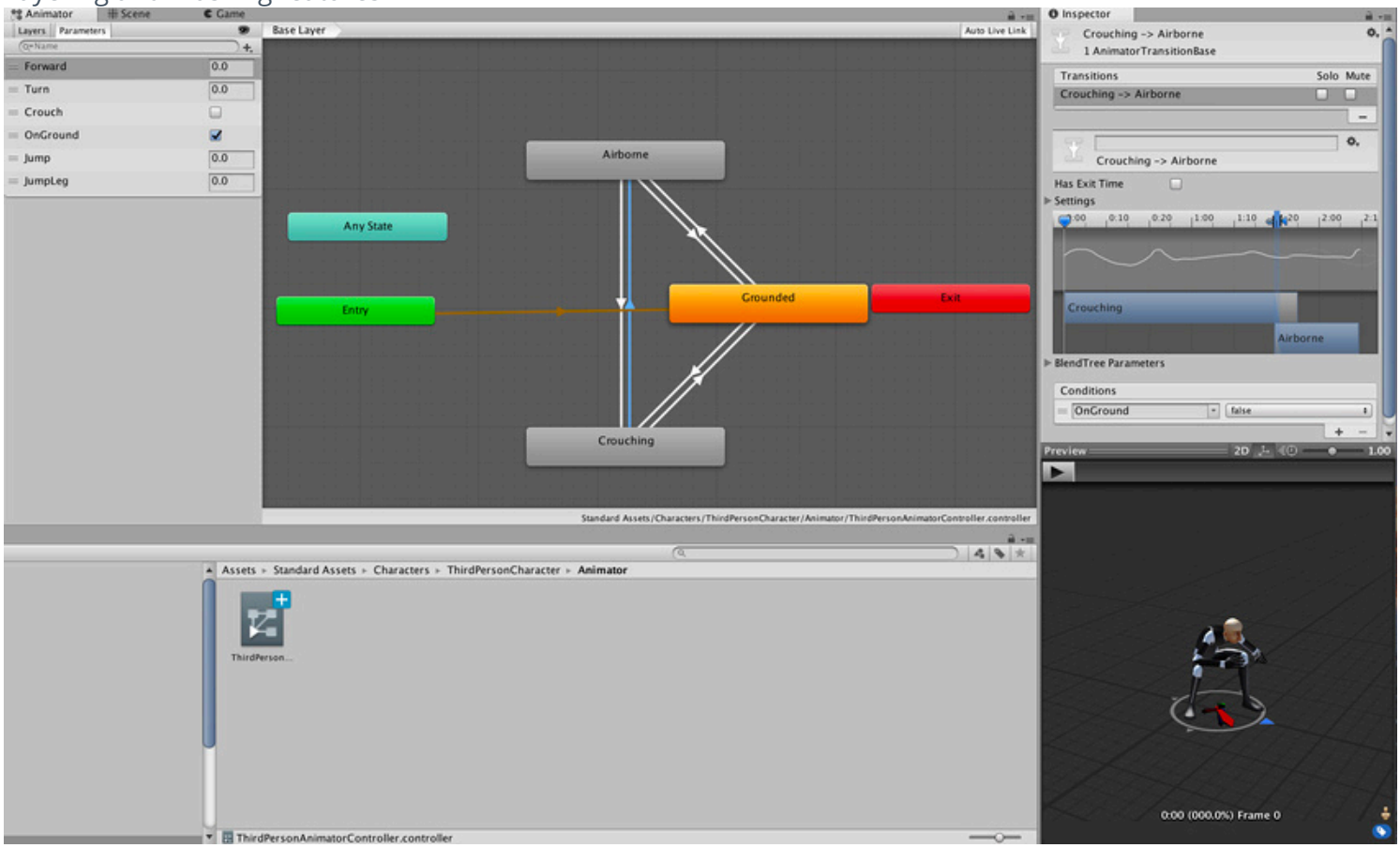
See the Knowledge Base Animation section for tips, tricks and troubleshooting.

Animation System Overview

Leave feedback

Unity has a rich and sophisticated animation system (sometimes referred to as '**Mecanim**'). It provides:

Easy workflow and setup of animations for all elements of Unity including objects, characters, and properties.
Support for imported animation clips and animation created within Unity
Humanoid animation retargeting - the ability to apply animations from one character model onto another.
Simplified workflow for aligning animation clips.
Convenient preview of animation clips, transitions and interactions between them. This allows animators to work more independently of programmers, prototype and preview their animations before gameplay code is hooked in.
Management of complex interactions between animations with a visual programming tool.
Animating different body parts with different logic.
Layering and masking features



Typical view of an Animation State Machine in the Animator window

Animation workflow

Unity's animation system is based on the concept of Animation Clips, which contain information about how certain objects should change their position, rotation, or other properties over time. Each clip can be thought of as a single linear recording. Animation clips from external sources are created by artists or animators with 3rd party tools such as Autodesk® 3ds Max® or Autodesk® Maya®, or come from motion capture studios or other sources.

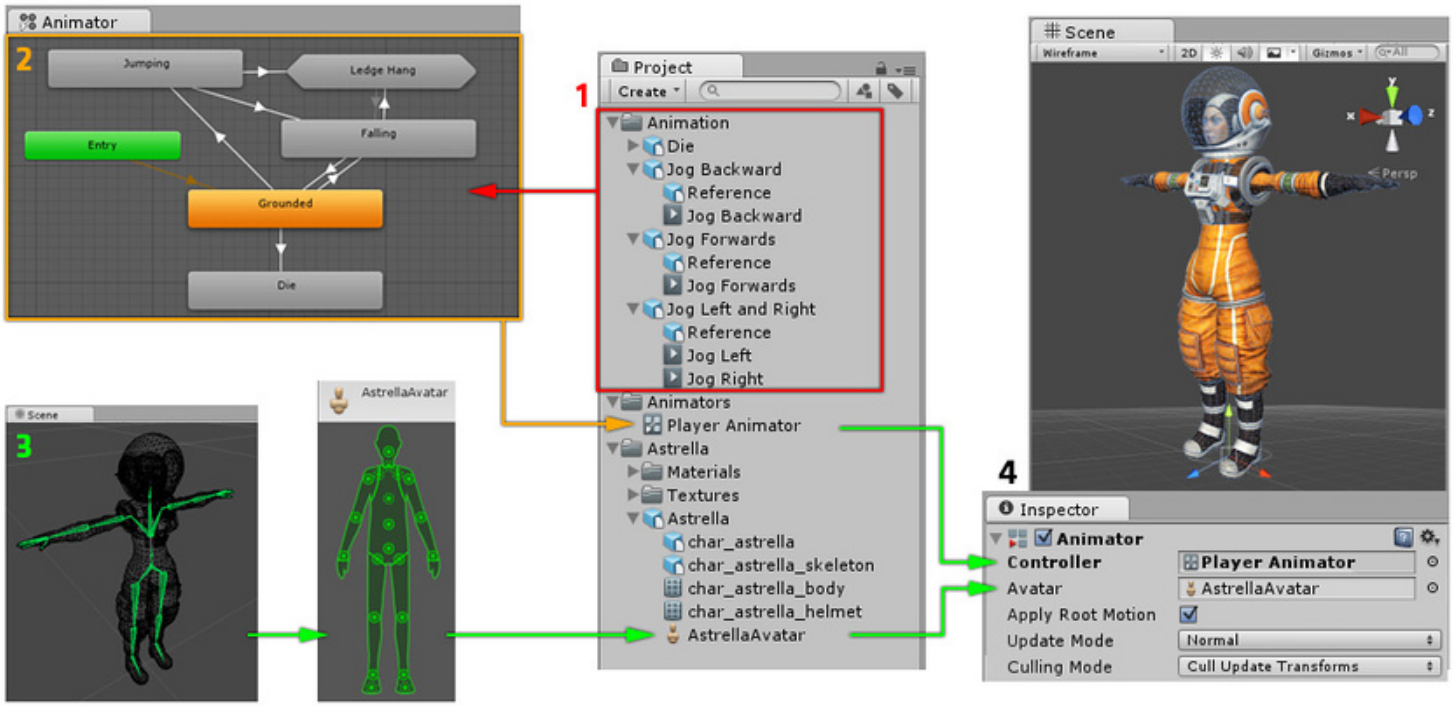
Animation Clips are then organised into a structured flowchart-like system called an Animator Controller. The Animator Controller acts as a "State Machine" which keeps track of which clip should currently be playing, and when the animations should change or blend together.

A very simple Animator Controller might only contain one or two clips, for example to control a powerup spinning and bouncing, or to animate a door opening and closing at the correct time. A more advanced Animator Controller might contain dozens of

humanoid animations for all the main character's actions, and might blend between multiple clips at the same time to provide a fluid motion as the player moves around the **scene**.

Unity's Animation system also has numerous special features for handling humanoid characters which give you the ability to retarget humanoid animation from any source (for example: motion capture; the Asset Store; or some other third-party animation library) to your own character model, as well as adjusting muscle definitions. These special features are enabled by Unity's Avatar system, where humanoid characters are mapped to a common internal format.

Each of these pieces - the Animation Clips, the Animator Controller, and the Avatar, are brought together on a **GameObject** via the Animator Component. This **component** has a reference to an Animator Controller, and (if required) the Avatar for this model. The Animator Controller, in turn, contains the references to the Animation Clips it uses.



How the various parts of the animation system connect together

The above diagram shows the following:

Animation clips are imported from an external source or created within Unity. In this example, they are imported motion captured humanoid animations.

The animation clips are placed and arranged in an Animator Controller. This shows a view of an Animator Controller in the **Animator window**. The States (which may represent animations or nested sub-state machines) appear as nodes connected by lines. This Animator Controller exists as an **asset** in the Project window.

The rigged character model (in this case, the astronaut "Astrella") has a specific configuration of bones which are mapped to Unity's common Avatar format. This mapping is stored as an Avatar asset as part of the imported character model, and also appears in the Project window as shown.

When animating the character model, it has an Animator component attached. In the **Inspector** view shown above, you can see the Animator Component which has both the Animator Controller and the Avatar assigned. The animator uses these together to animate the model. The Avatar reference is only necessary when animating a humanoid character. For other types of animation, only an Animator Controller is required.

Unity's animation system comes with a lot of concepts and terminology. If at any point, you need to find out what something means, go to our Animation Glossary.

Legacy animation system

While Mecanim is recommended for use in most situations, Unity has retained its legacy animation system which existed before Unity 4. You may need to use when working with older content created before Unity 4. For information on the Legacy animation

system, see this section

2018–04–25 Page amended with limited editorial review

Animation Clips

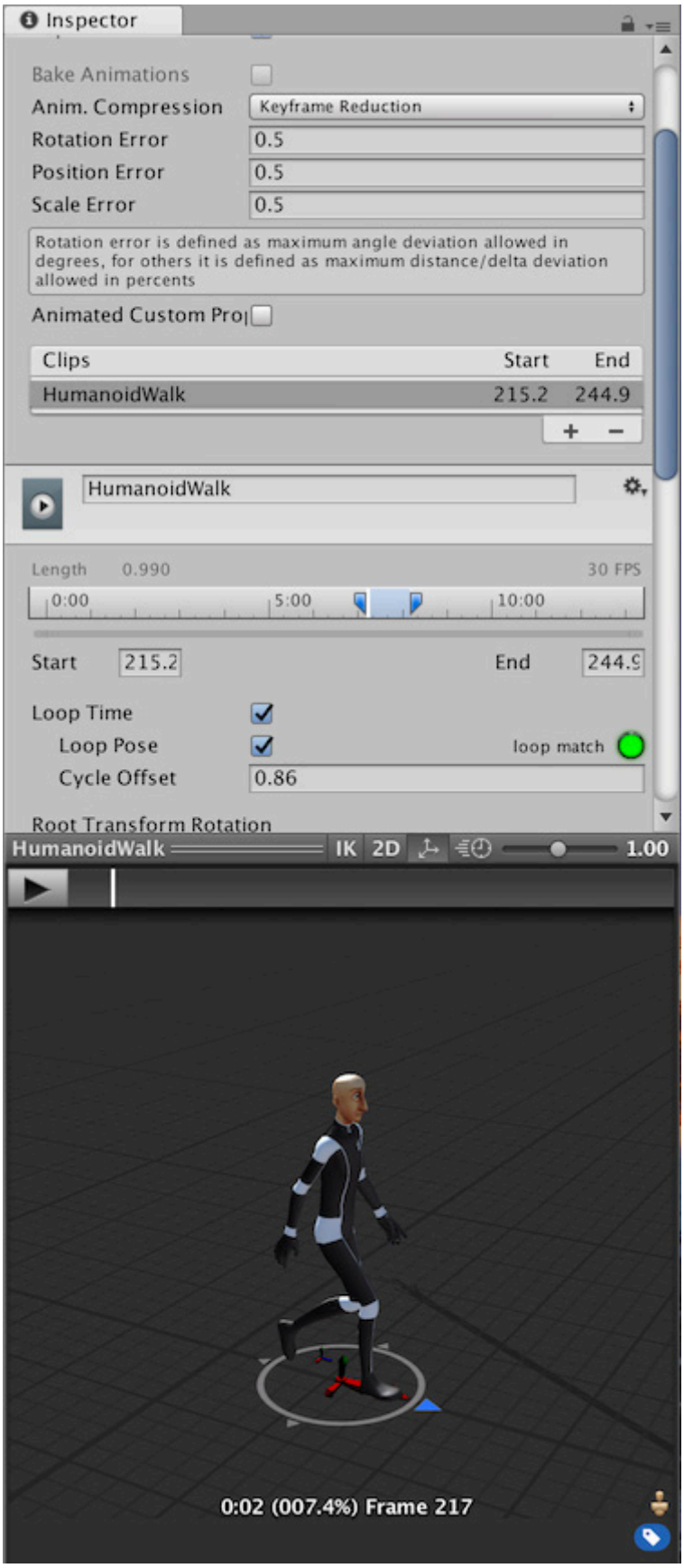
Leave feedback

Animation Clips are one of the core elements to Unity's animation system. Unity supports importing animation from external sources, and offers the ability to create **animation clips** from scratch within the editor using the Animation window.

Animation from External Sources

Animation clips imported from external sources could include:

Humanoid animations captured at a motion capture studio
Animations created from scratch by an artist in an external 3D application (such as Autodesk® 3ds Max® or Autodesk® Maya®)
Animation sets from 3rd-party libraries (eg, from Unity's asset store)
Multiple clips cut and sliced from a single imported **timeline**.

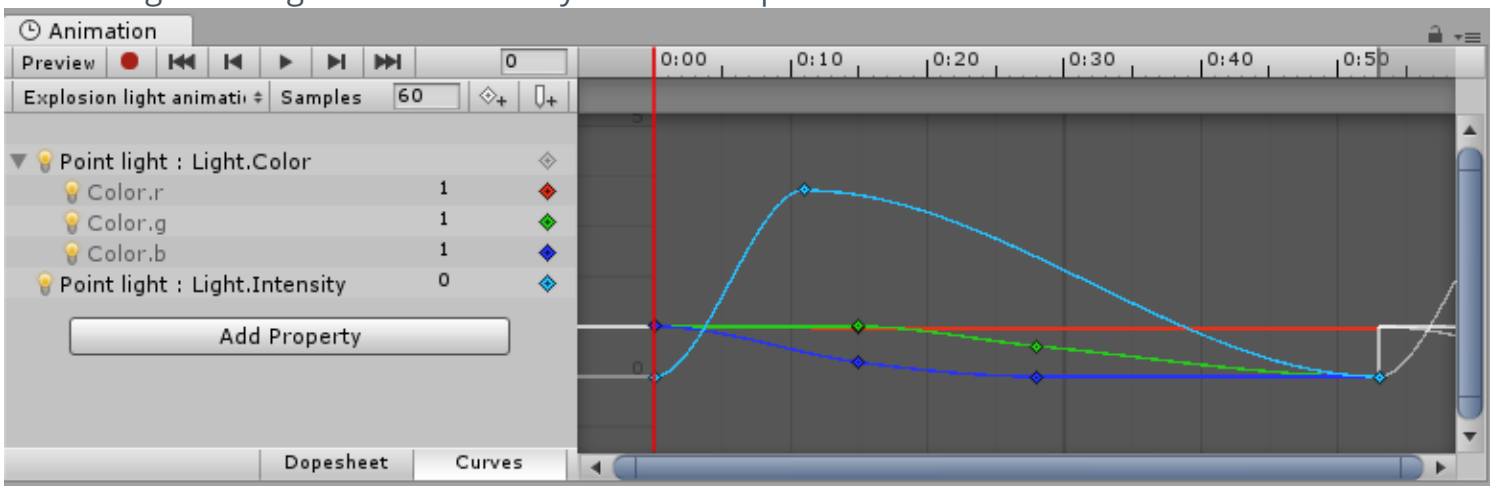


An example of an imported animation clip, viewed in Unity's Inspector window

Animation Created and Edited Within Unity

Unity's Animation Window also allows you to create and edit animation clips. These clips can animate:

The position, rotation and scale of GameObjects
Component properties such as material colour, the intensity of a light, the volume of a sound
Properties within your own **scripts** including float, integer, enum, vector and Boolean variables
The timing of calling functions within your own scripts



An example of Unity's Animation window being used to animate parameters of a component - in this case, the intensity and range of a point light

---

2017–10–02 Page amended with limited editorial review

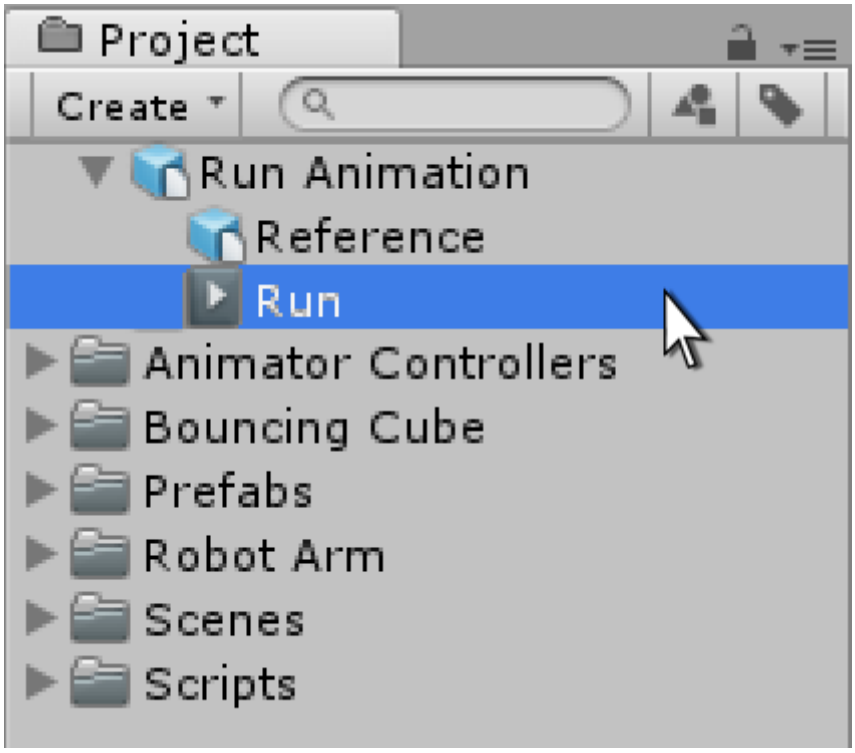
Integer properties within scripts can be animated in 2017.31

2D Button added to animation previewer 2017.3

Animation from external sources

Leave feedback

Animation from external sources is imported into Unity in the same way as regular 3D files. These files, whether they're generic FBX files or native formats from 3D software such as Autodesk® Maya®, Cinema 4D, Autodesk® 3ds Max®, can contain animation data in the form of a linear recording of the movements of objects within the file.



An imported FBX 3D Asset containing an animation titled 'Run'

In some situations the object to be animated (eg, a character) and the animations to go with it can be present in the same file. In other cases, the animations may exist in a separate file to the model to be animated.

It may be that animations are specific to a particular model, and cannot be re-used on other models. For example, a giant octopus end-boss in your game might have a unique arrangement of limbs and bones, and its own set of animations.

In other situations, it may be that you have a library of animations which are to be used on various different models in your **scene**. For example, a number of different humanoid characters might all use the same walk and run animations. In these situations, it's common to have a simple placeholder model in your animation files for the purposes of previewing them. Alternatively, it is possible to use animation files even if they have no geometry at all, just the animation data.

When importing multiple animations, the animations can each exist as separate files within your project folder, or you can extract multiple **animation clips** from a single FBX file if exported as takes from Motion builder or with a **plugin** / script for Autodesk® Maya®, Autodesk® 3ds Max® or other 3D packages. You might want to do this if your file contains multiple separate animations arranged on a single **timeline**. For example, a long motion captured **timeline** might contain the animation for a few different jump motions, and you may want to cut out certain sections of this to use as individual clips and discard the rest. Unity provides animation cutting tools to achieve this when you import all animations in one **timeline** by allowing you to select the frame range for each clip.

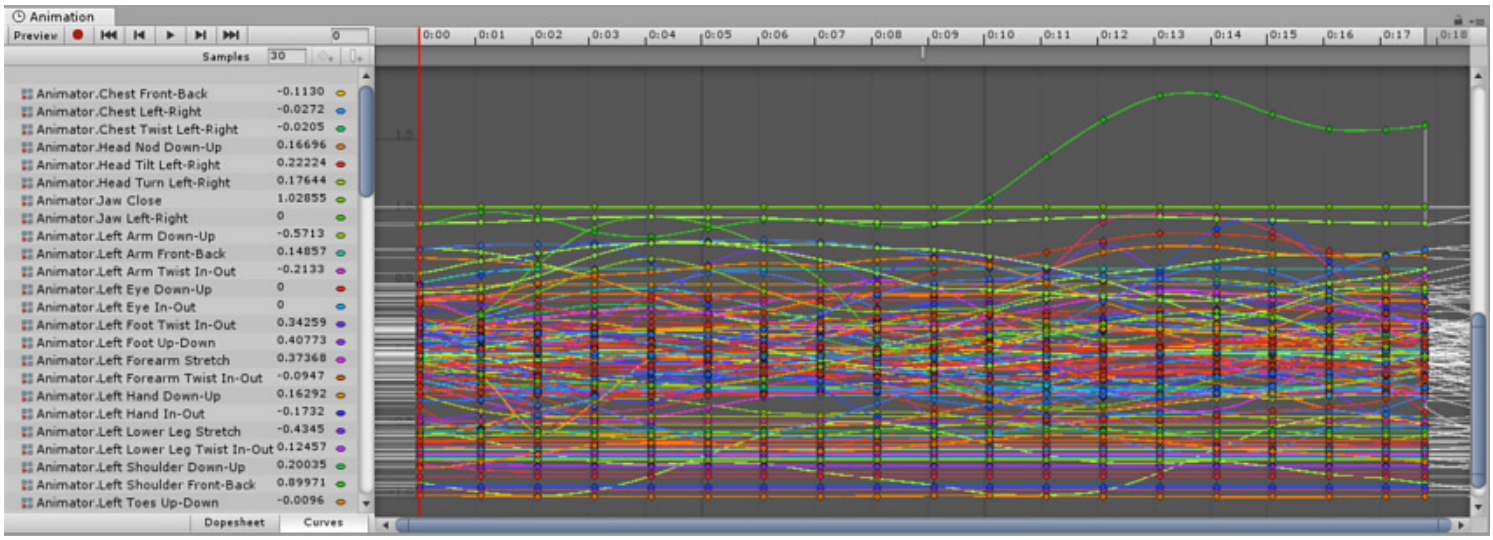
Importing animation files

Before any animation can be used in Unity, it must first be imported into your project. Unity can import native Autodesk® Maya® (.mb or .ma), Autodesk® 3ds Max® (.max) and Cinema 4D (.c4d) files, and also generic FBX files which can be exported from most animation packages.

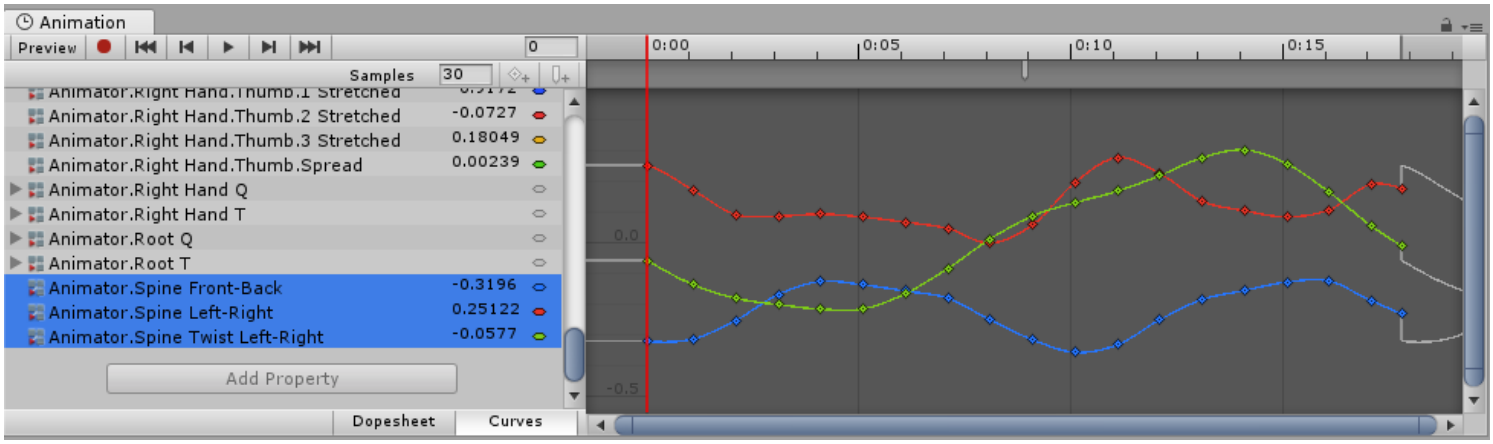
For more information, see Importing.

Viewing and copying data from imported animation files

You can view the **keyframes** and curves of imported animation clips in the Animation window. Sometimes, if these imported clips have lots of bones with lots of **keyframes**, the amount of information can look overwhelmingly complex. For example, the image below is what a humanoid running animation looks like in the Animation window:

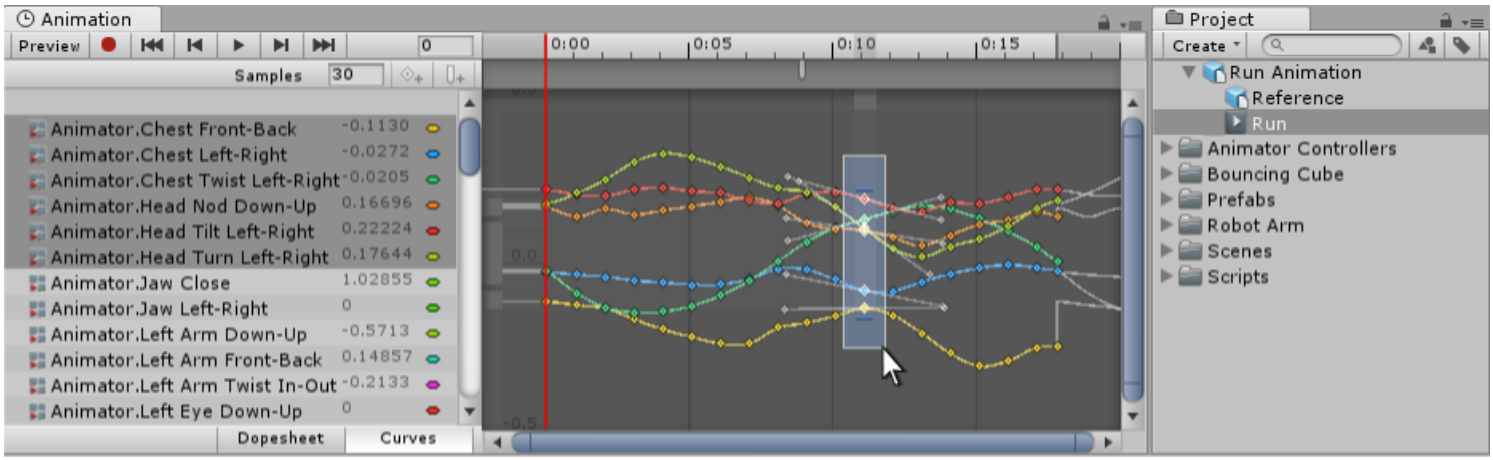


To simplify the view, select the specific bones you are interested in examining. The Animation window then displays only the keyframes or curves for those bones.



Limiting the view to just the selected bones

When viewing imported Animation keyframes, the Animation window provides a read-only view of the Animation data. To edit this data, create a new empty Animation Clip in Unity (see Creating a new Animation Clip), then select, copy and paste the Animation data from the imported Animation Clip into your new, writable Animation Clip.



Selecting keyframes from an imported clip.

2018–04–25 Page amended with limited editorial review

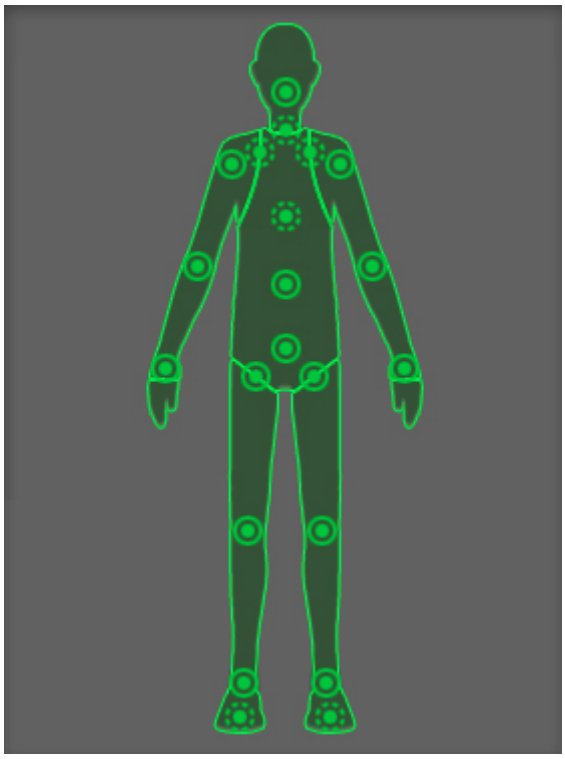
Humanoid Avatars

Leave feedback

Unity's Animation System has special features for working with humanoid characters. Because humanoid characters are so common in games, Unity provides a specialized workflow, and an extended tool set for **humanoid animations**.

The Avatar system is how Unity identifies that a particular animated model is humanoid in layout, and which parts of the model correspond to the legs, arms, head and body.

Because of the similarity in bone structure between different humanoid characters, it is possible to map animations from one humanoid character to another, allowing **retargeting** and **inverse** kinematics__ (IK)__.



Unity's Avatar structure

2018–04–25 Page amended with limited editorial review

Animation Window Guide

Leave feedback

The **Animation Window** in Unity allows you to create and modify **Animation Clips** directly inside Unity. It is designed to act as a powerful and straightforward alternative to external 3D animation programs. In addition to animating movement, the editor also allows you to animate variables of materials and **components** and augment your Animation Clips with **Animation Events**, functions that are called at specified points along the **timeline**.

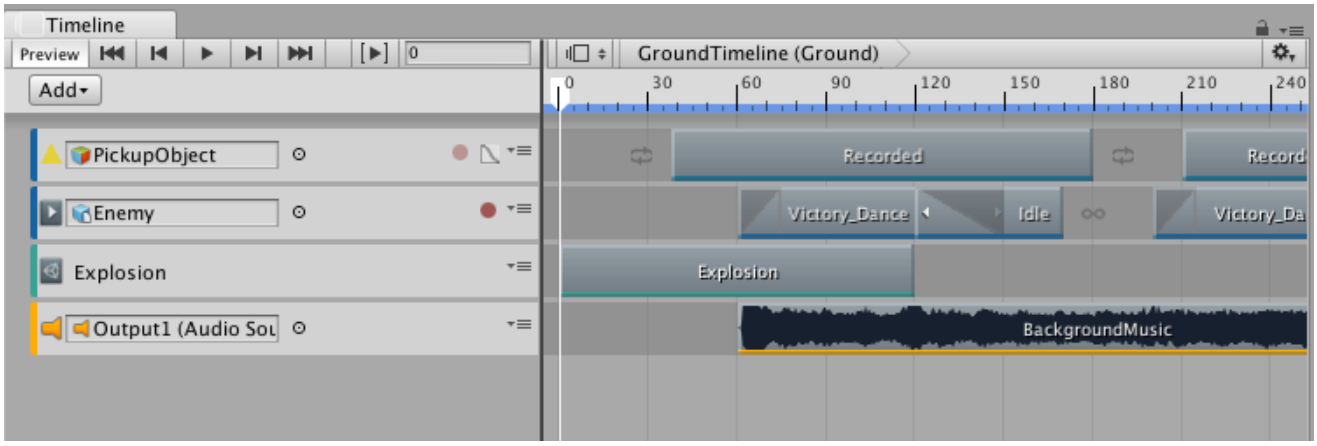
See the pages about Animation import and Animation Scripting for further information about these subjects.

What's the difference between the Animation window and the Timeline window?

The Timeline window

The Timeline window allows you to create cinematic content, game-play sequences, audio sequences and complex particle effects. You can animate many different **GameObjects** within the same sequence, such as a cut **scene** or scripted sequence where a character interacts with scenery. In the timeline window you can have multiple types of track, and each track can contain multiple clips that can be moved, trimmed, and blended between. It is useful for creating more complex animated sequences that require many different GameObjects to be choreographed together.

The Timeline window is newer than the Animation window. It was added to Unity in version 2017.1, and supercedes some of the functionality of the Animation window. To start learning about Timeline in Unity, visit the Timeline section of the user manual.



The Timeline window, showing many different types of clips arranged in the same sequence

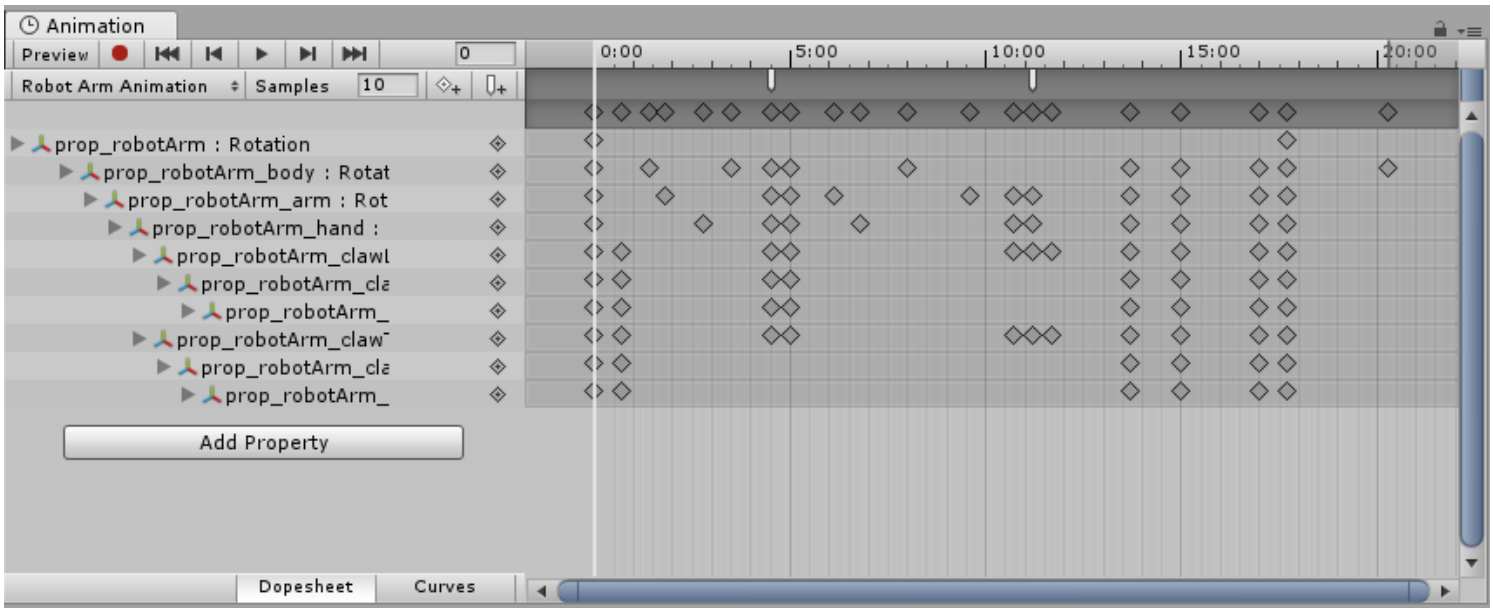
The Animation window

The Animation window allows you to create individual animation clips as well as viewing imported animation clips. Animation clips store animation for a single GameObject or a single hierarchy of GameObjects. The Animation window is useful for animating discrete items in your game such as a swinging pendulum, a sliding door, or a spinning coin. The animation window can only show one animation clip at a time.

The Animation window was added to Unity in version 4.0. The Animation window is an older feature than the Timeline window. It provides a simple way to create animation clips and animate individual GameObjects, and the clips you create in the Animation window can be combined and blended between using anim Animator controller. However, to create more complex sequences involving many disparate GameObjects you should use the Timeline window (see above).

The animation window has a "timeline" as part of its user interface (the horiontal bar with time delineations marked out), however this is separate to the Timeline window.

To start learning about animation in Unity, visit the Animation section of the user manual.



The Animation window, shown in dopesheet mode, showing a hierarchy of objects (in this case, a robot arm with numerous moving parts) animated together in a single animation clip

Using the Animation view

Leave feedback

The **Animation view** is used to preview and edit **Animation Clips** for animated **GameObjects** in Unity. To open the Animation view in Unity, go to **Window** > **Animation**.

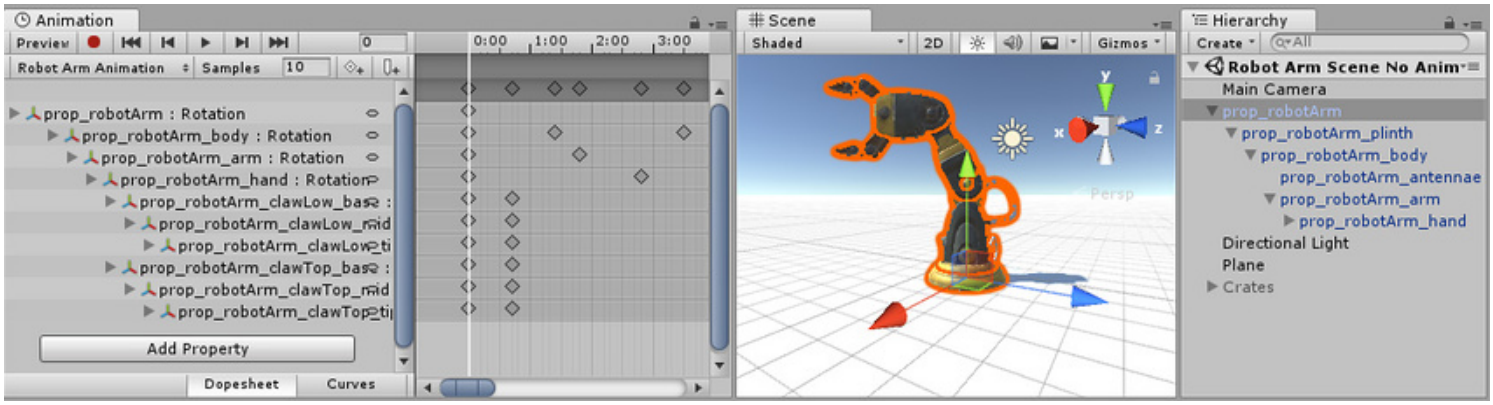
Viewing Animations on a GameObject

The **Animation window** is linked with with the Hierarchy window, the Project window, the **Scene view**, and the **Inspector** window. Like the **Inspector**, the Animation window shows the **timeline** and keyframes of the Animation for the currently selected GameObject or Animation Clip **Asset**. You can select a GameObject using the Hierarchy window or the **Scene View**, or select an Animation Clip **Asset** using the Project Window.

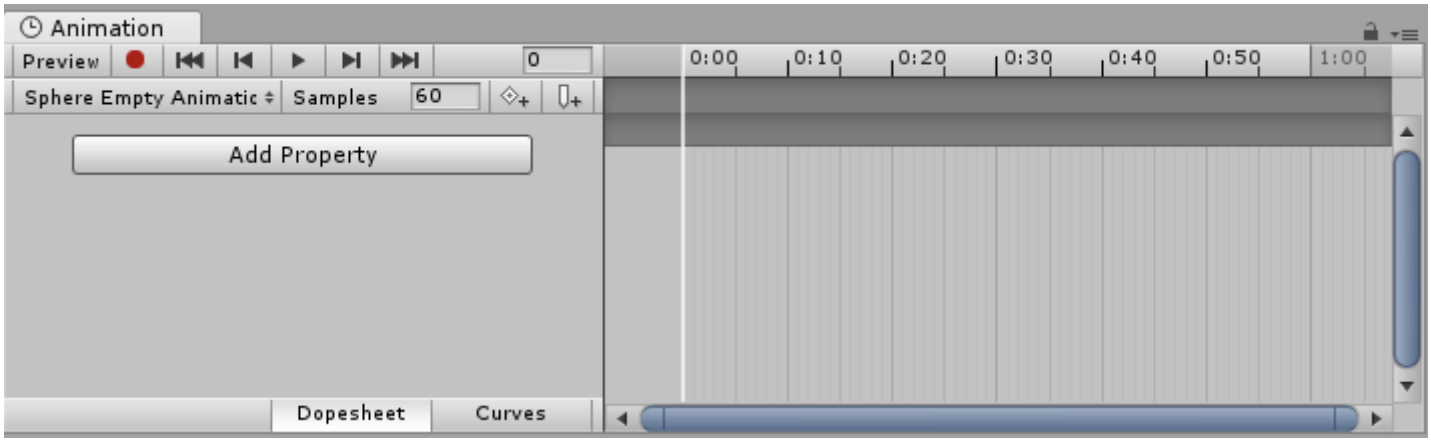
Note: The Animation view is separate from, but looks similar to the Timeline window.

The Animated Properties list

In the image below, the Animation view (left) shows the Animation used by the currently selected GameObject, and its child GameObjects if they are also controlled by this Animation. The **Scene** view and Hierarchy view are on the right, demonstrating that the Animation view shows the Animations attached to the currently selected GameObject.



In the left side of the Animation view is a list of the animated properties. In a newly created clip where no animation has yet been recorded, this list is empty.



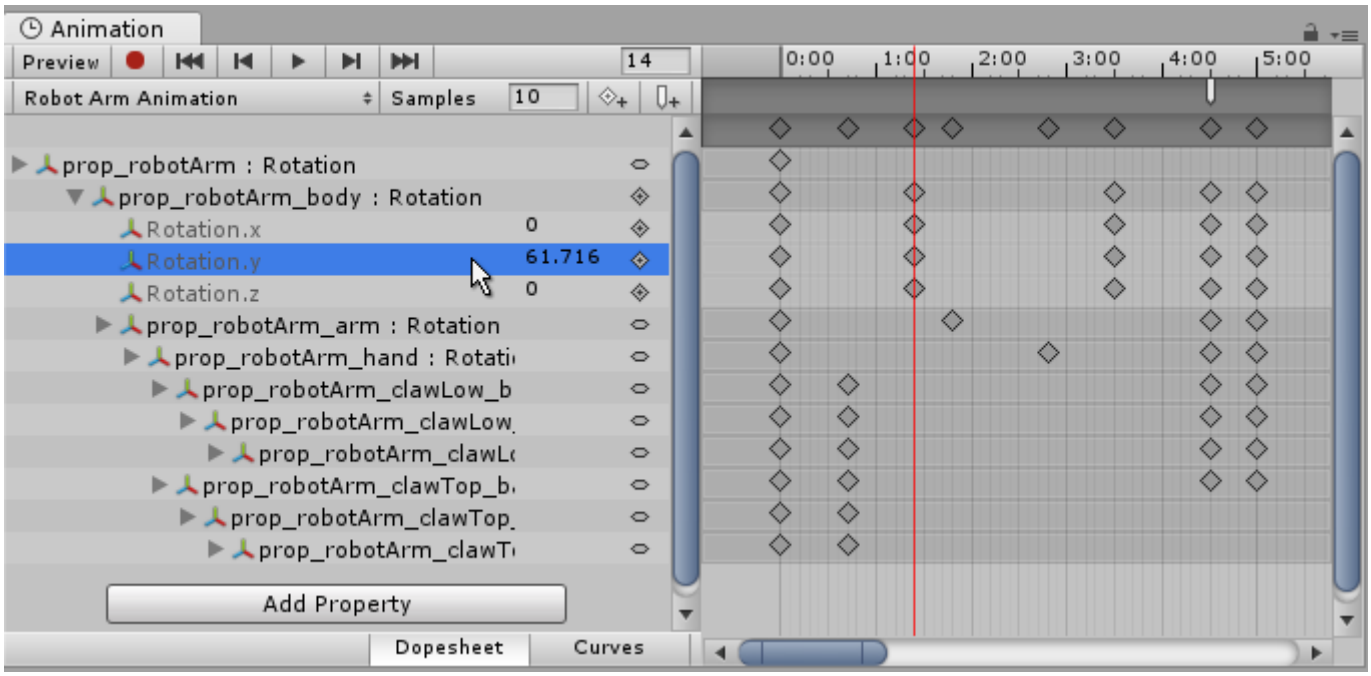
The Animation view displaying an empty clip. No properties are shown on the left yet.

When you begin to animate various properties within this clip, the animated properties will appear here. If the animation controls multiple child objects, the list will also include hierarchical sub-lists of each child object's

animated properties. In the example above, various parts of the Robot Arm's GameObject hierarchy are all animated within the same animation clip.

When animating a hierarchy of GameObjects within a single clip like this, make sure you create the Animation on the root GameObject in the hierarchy.

Each property can be folded and unfolded to reveal the exact values recorded at each keyframe. The value fields show the interpolated value if the playback head (the white line) is between keyframes. You can edit these fields directly. If changes are made when the playback head is over a keyframe, the keyframe's values are modified. If changes are made when the playback head is between keyframes (and therefore the value shown is an interpolated value), a new keyframe is created at that point with the new value that you entered.



An unfolded property in the Animation View, allowing the keyframe value to be typed in directly. In this image, an interpolated value is shown because the playback head (the white line) is between keyframes. Entering a new value at this point would create a new keyframe.

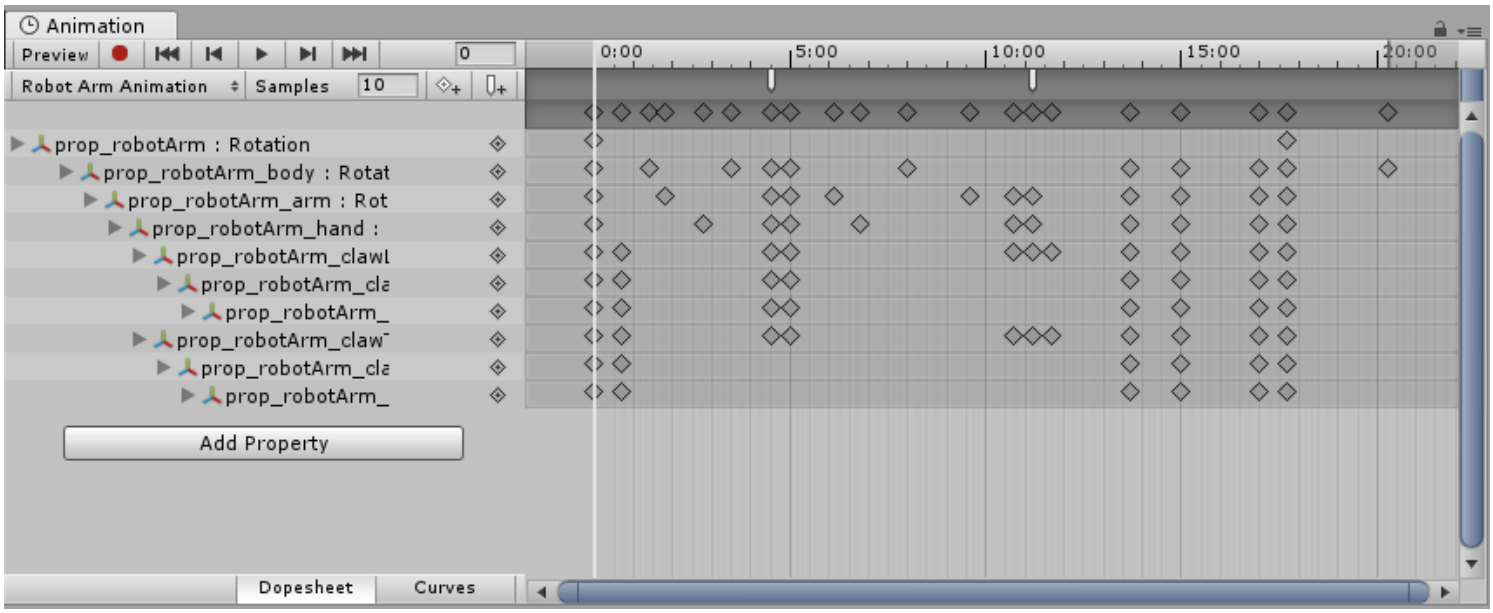
The Animation Timeline

On the right side of the Animation View is the timeline for the current clip. The keyframes for each animated property appear in this timeline. The timeline view has two modes, **Dopesheet** and **Curves**. To toggle between these modes, click **Dopesheet** or **Curve** at the bottom of the animated property list area:

These offer two alternate views of the Animation timeline and keyframe data.

Dopesheet timeline mode

Dopesheet mode offers a more compact view, allowing you to view each property's keyframe sequence in an individual horizontal track. This allows you to view a simple overview of the keyframe timing for multiple properties or GameObjects.

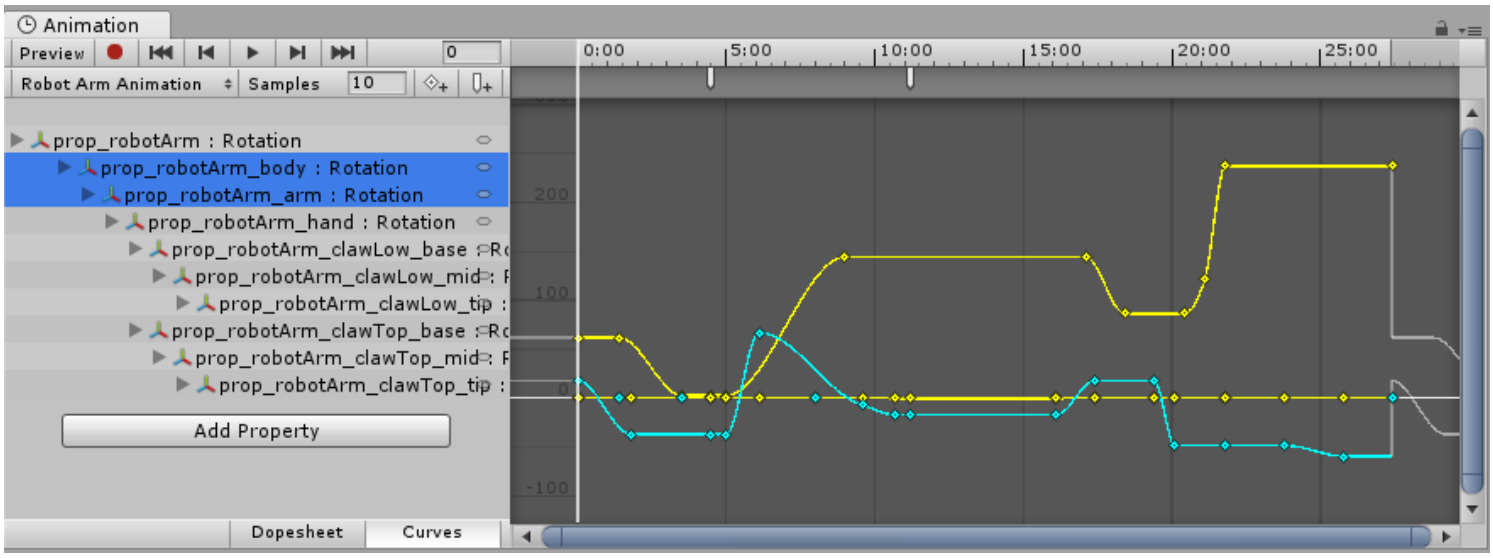


Here the Animation Window is in Dope Sheet mode, showing the keyframe positions of all animated properties within the Animation clip

See documentation on Key manipulation in Dopesheet mode for more information.

Curves timeline mode

Curves mode displays a resizable graph containing a view of how the values for each animated property changes over time. All selected properties appear overlaid within the same graph view. This mode allows you to have great control over viewing and editing the values, and how they are interpolated between.

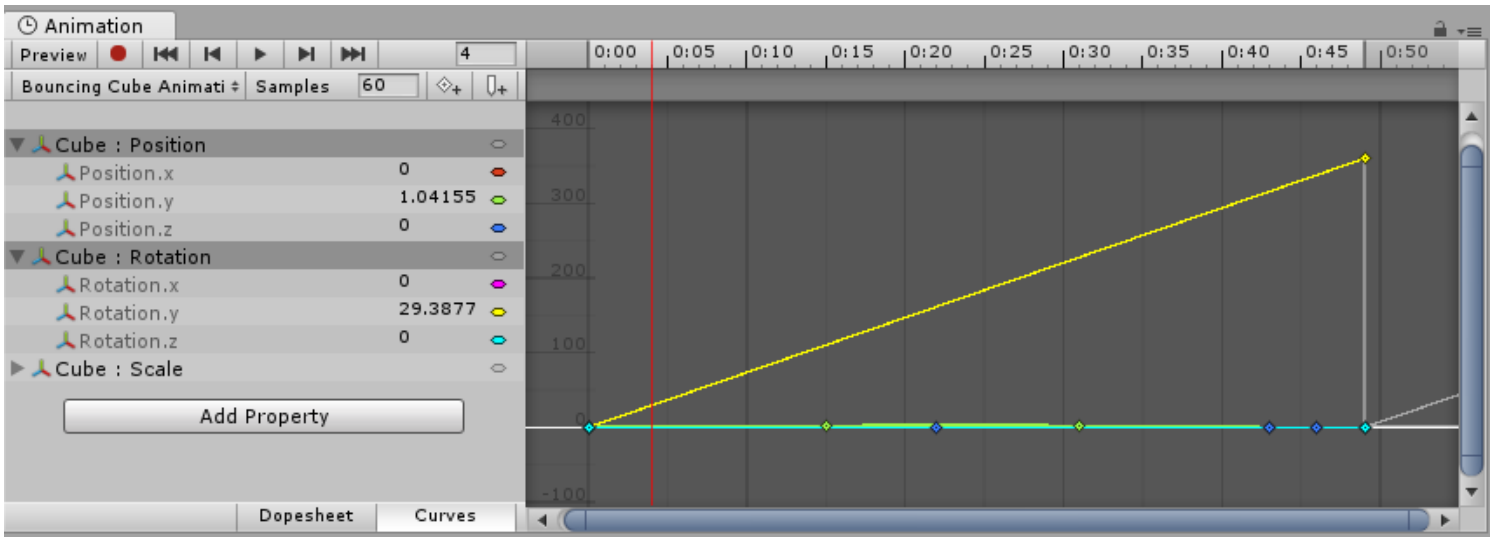


Here, the Animation Window shows the curves for the rotation data of four selected GameObjects within this Animation clip

Fitting your selection to the window

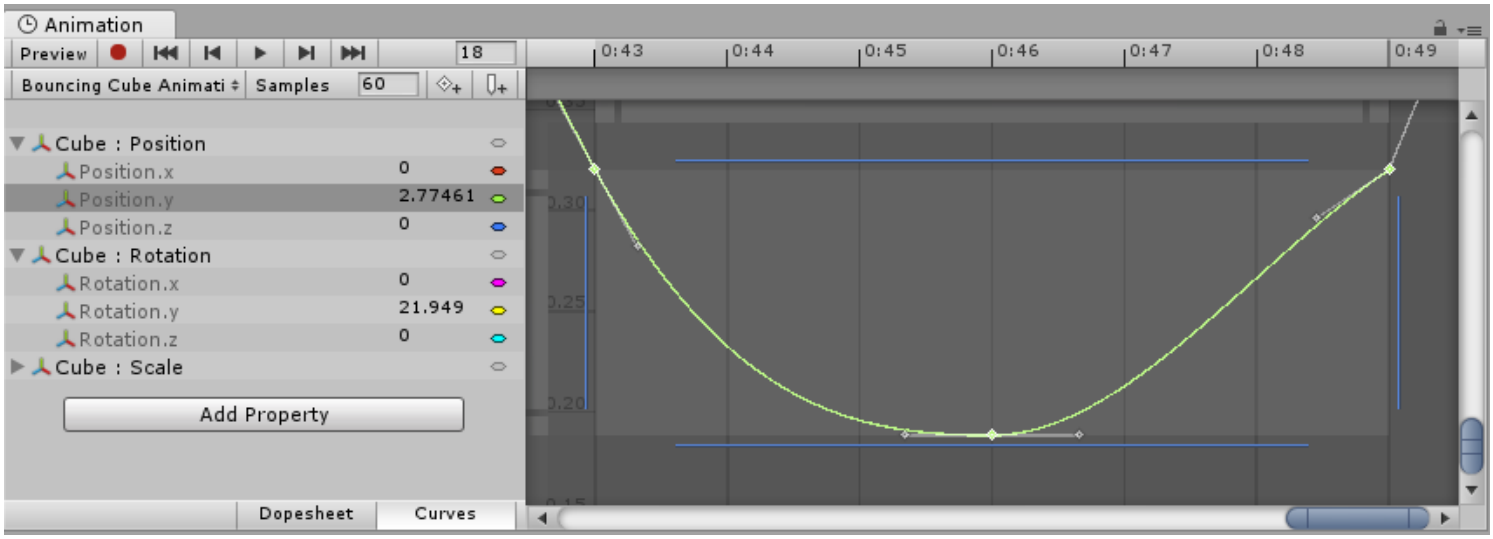
When using **Curves** mode to view your Animation, it's important to understand that sometimes the various ranges for each property can differ greatly. For example, consider a simple Animation clip for a spinning bouncing cube. The bouncing Y position value may vary between the range 0 to 2 (meaning the cube bounces 2 units high during the animation); however, the rotation value goes from 0 to 360 (representing its degrees of rotation). When viewing these two curves at the same time, the **animation curves** for the position values will be very

difficult to make out because the view will be zoomed out to fit the 0–360 range of the rotation values within the window:

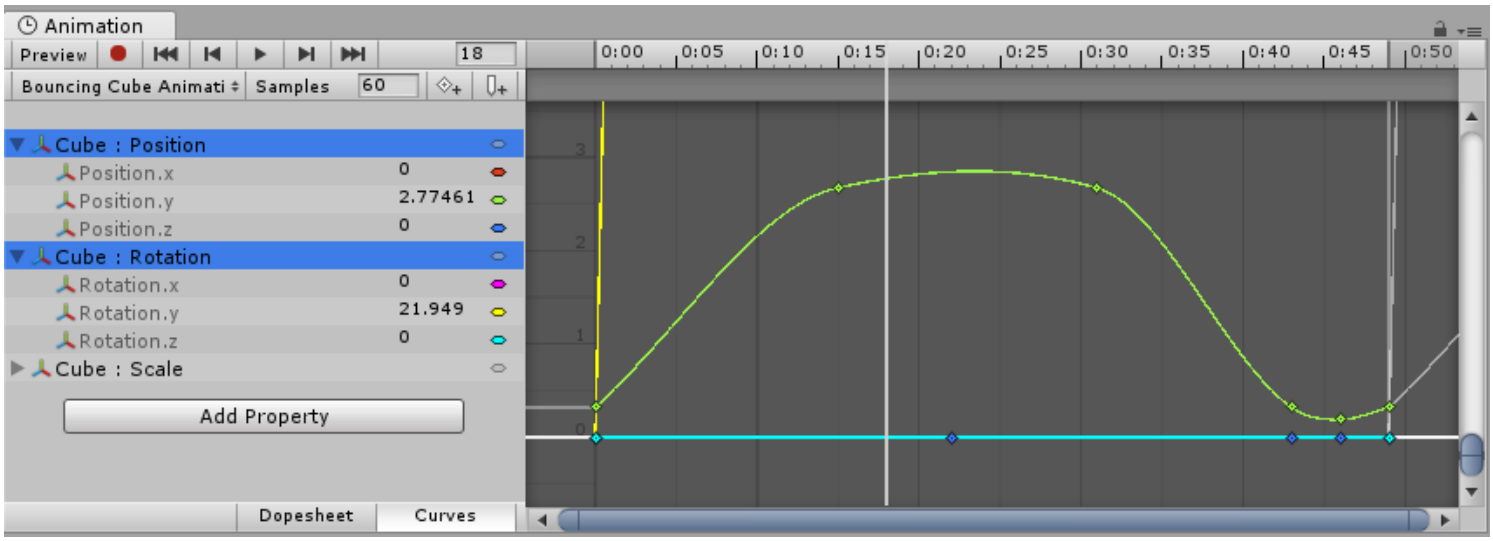


The position and rotation curves of a bouncing spinning cube are both selected, but because the view is zoomed out to fit the 0–360 range of the rotation curve, the bouncing Y position curve is not easily discernible

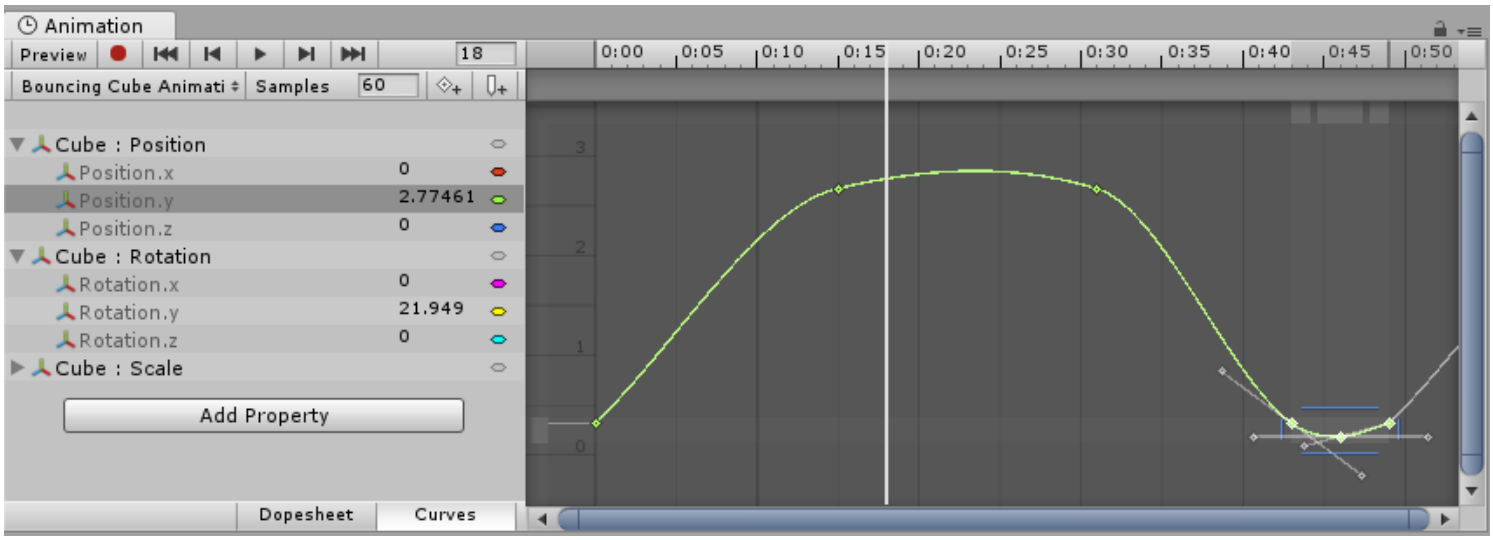
Press **F** on the keyboard to zoom the view to the currently selected keyframes. This is useful as a quick way to focus and re-scale the window on a portion of your Animation timeline for easier editing.



Click on individual properties in the list and press **F** on the keyboard to automatically re-scale the view to fit the range for that value. You can also manually adjust the zoom of the **Curves** window by using the drag handles at each end of the view's scrollbar sliders. In the image below, the Animation Window is zoomed in to view the bouncing Y position Animation. The start of the yellow rotation curve is still visible, but now extends way off the top of the view:



Press **A** on the keyboard to fit and re-scale the window to show all the keyframes in the clip, regardless of which ones are selected. This is useful if you want to view the whole timeline while preserving your current selection:



Playback and frame navigation controls

To control playback of the **Animation Clip**, use the **Playback Controls** at the top left of Animation view.

Frame navigation

From left-to-right, these controls are:

Preview mode (toggle on/off)
Record mode (toggle on/off) *Note: Preview mode is always on if record mode is on*
Move playback head to the beginning of the clip
Move playback head to the previous keyframe
Play Animation
Move playback head to the next keyframe
Move playback head to the end of the clip

You can also control the playback head using the following keyboard shortcuts:

Press **Comma** (**,**) to go to the previous frame.
Press **Period** (**.**) to go to the next frame.

Hold Alt and press **Comma** (**,**) to go to the previous **keyframe**.
Hold Alt and press **Period** (**.**) to go to the next **keyframe**.

Locking the window

You can lock the Animation editor window so that it does not automatically switch to reflect the currently selected GameObject in the Hierarchy or Scene. Locking the window is useful if you want to focus on the Animation for one particular GameObject, and still be able to select and manipulate other GameObjects in the Scene.

The Lock button

To learn more about navigating the Curve view, see documentation on Using Animation Curves.

2017–09–05 Page amended with limited editorial review

Creating a New Animation Clip

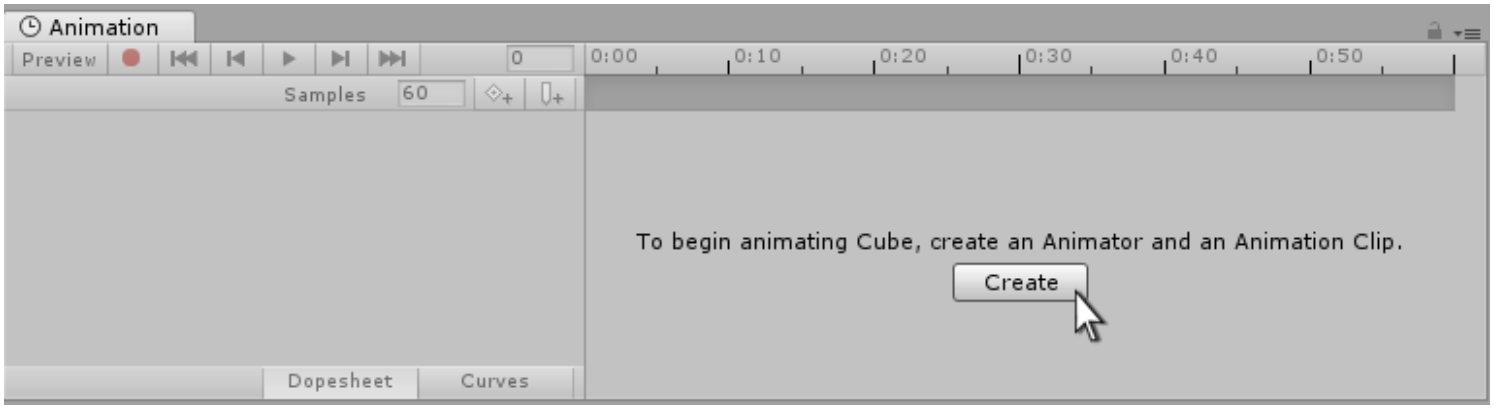
Leave feedback

To animate **GameObjects** in Unity, the object or objects need an **Animator Component** attached. This Animator **Component** must reference an **Animator Controller**, which in turn contains references to one or more **Animation Clips**.

When using the Animation View to begin animating a GameObject in Unity, all these items will be automatically created, attached and set-up for you.

To create a new **Animation Clip** for the selected GameObject, and make sure the **Animation Window** is visible.

If the GameObject does not yet have any Animation Clips assigned, you will see the “Create” button in the centre of the Animation Window **timeline** area. Click the Create button. You will then be prompted to save your new empty Animation Clip somewhere in your **Assets** folder.



Create a new Animation Clip

Once you have saved this new empty Animation Clip, a number of things happen automatically:

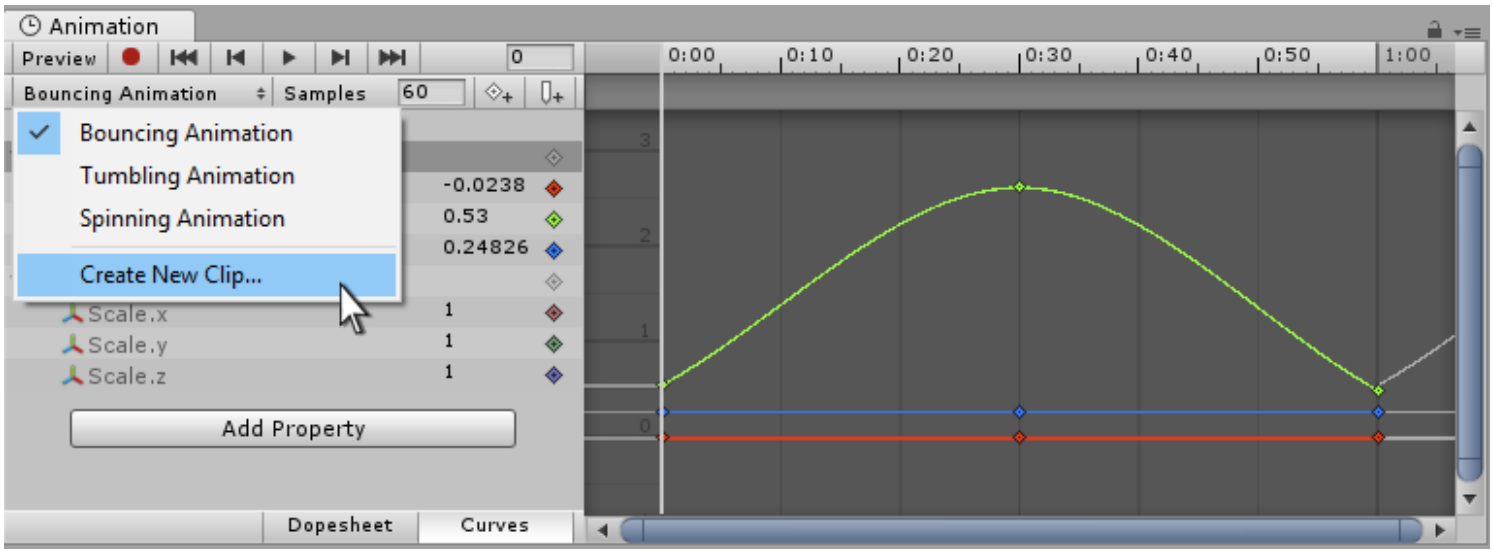
A new Animator Controller asset will be created
The new clip being created will be added into the Animator Controller as the default state
An Animator Component will be added to the GameObject being animated
The Animator Component will have the new Animator Controller assigned to it

The result of this automatic sequence is that all the required elements of the animation system are set up for you, and you can now begin animating the objects.

Adding another Animation Clip

If the Game Object already has one or more Animation Clips assigned, the “Create” button will not be visible. Instead, one of the clips will be visible in the animation window. You can switch between which Animation Clip is visible in the window by using the menu in the top-left of the Animation window, just under the playback controls.

If you want to create a new Animation Clip on an object that already has animations, you must select “Create New Clip” from this menu. Again, you will be prompted to save your new empty Animation Clip before being able to work with it.

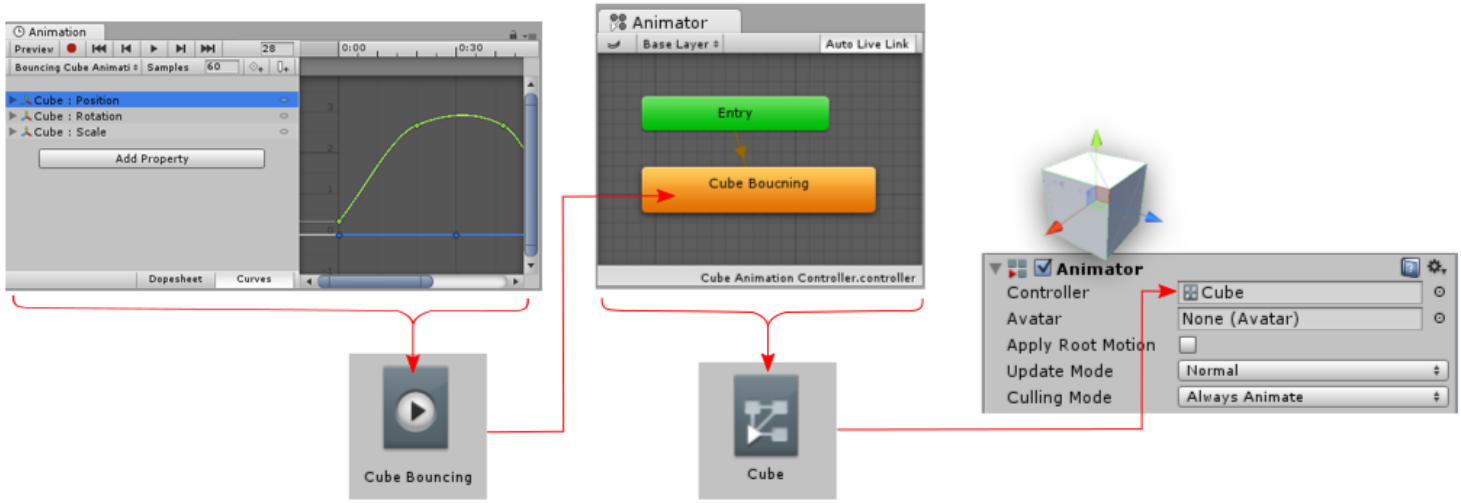


Adding an additional new Animation Clip to an object which already has some clips assigned

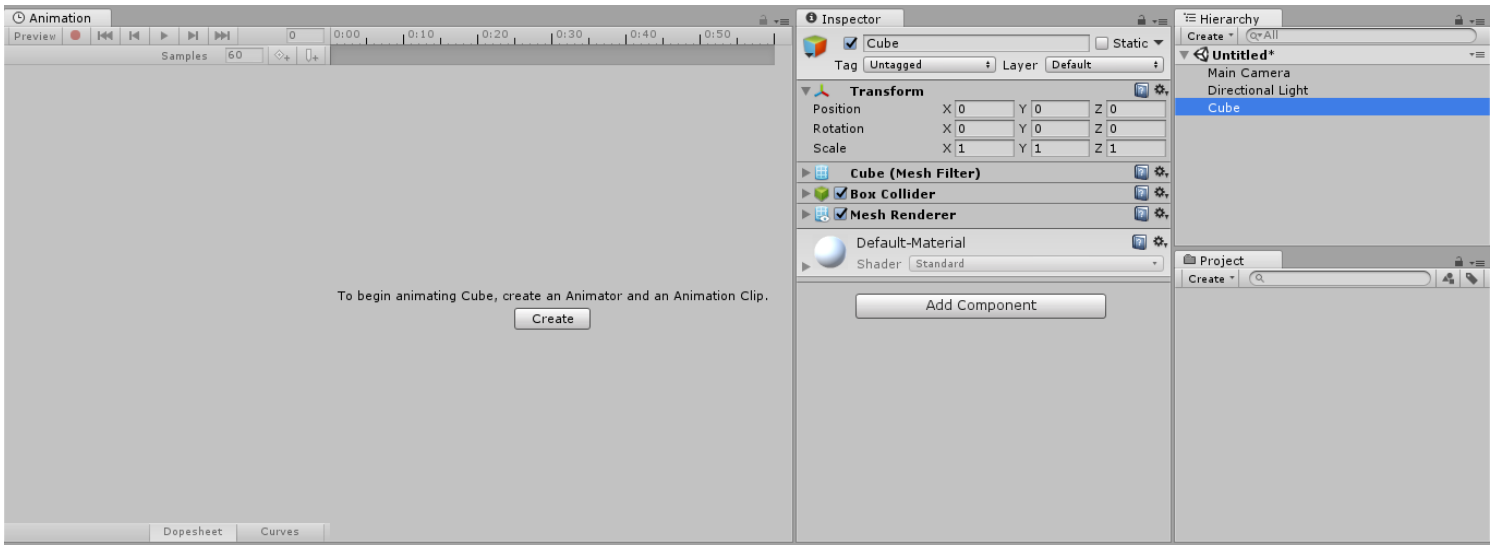
How it fits together

While the above steps automatically set up the relevant components and references, it can useful to understand which pieces must be connected together.

A GameObject must have an **Animator** component
The Animator component must have an **Animator Controller** asset assigned
The Animator Controller asset must have one or more Animation Clips assigned
The diagram below shows how these pieces are assigned, starting from the new animation clip created in the Animation Window:

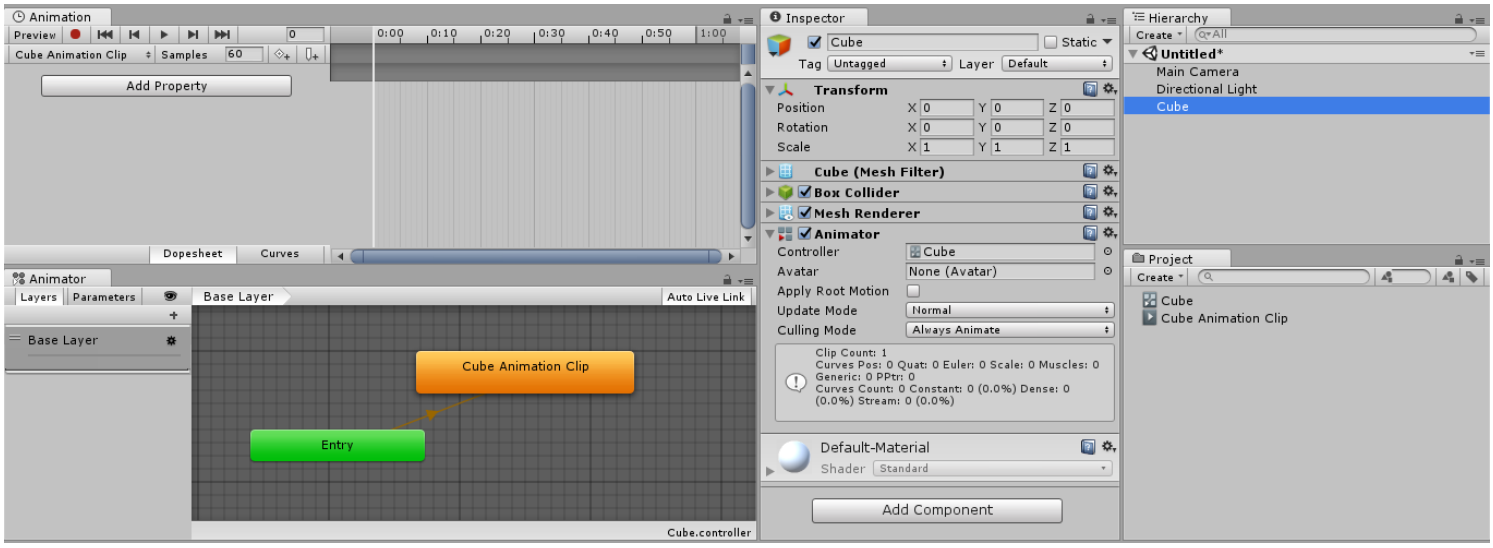


A new clip is created, and saved as an asset. The clip is automatically added as the default state to a new Animator Controller which is also saved as an asset. The Animator Controller is assigned to an Animator Component which is added to the GameObject.
In the image below, you can see a GameObject selected ("Cube") that is not yet animated. We have just a simple cube, with no Animator component. The Animation, Hierarchy, Project and **Inspector** windows are arranged side-by-side for clarity.



Before: An un-animated gameobject ("Cube") is selected. It does not yet have an Animator Component, and no Animator Controller exists.

By pressing the create button in the Animation view, a new animation clip is created. Unity will ask to pick the name & location to save this new Animation Clip. Unity also creates an Animator Controller asset with the same name as the selected GameObject, adds an Animator component to the GameObject, and connects the assets up appropriately.



After: After creating a new clip, you can see the new assets created in the project window, and the Animator Component assigned in the Inspector window (far right). You can also see the new clip assigned as the default state in the Animator Window

In the new view above, you can see:

The Animation Window (top left) now shows a timeline with a white playback head line, ready to record new **keyframes**. The clip's name is visible in the clip menu, just below the playback controls.
The Inspector (center) shows that the Cube GameObject now has an **Animator Component** added, and the "Controller" field of the component shows that an Animator Controller asset called "Cube" is assigned
The Project Window (bottom right) shows that two new assets have been created - An Animator Controller asset called "Cube" and an Animation Clip asset called "Cube Animation Clip"
The **Animator Window** (bottom left) shows the contents of the Animator Controller - you can see that the Cube Animation Clip has been added to the controller, and that it is the "default state" as

indicated by the orange color. Subsequent clips added to the controller would have a grey color, indicating they are not the default state.

2017–09–05 Page amended with limited editorial review

Animating a GameObject

Leave feedback

Once you have saved the new Animation clip **Asset**, you are ready to begin adding **keyframes** to the clip.

There are two distinct methods you can use to animate **GameObjects** in the Animation window: **Record Mode** and **Preview Mode**.

Record Mode. (Also referred to as auto-key mode)

The Animation Window in Record mode

In record mode, Unity automatically creates keyframes at the playback head when you move, rotate, or otherwise modify any **animatable property** on your animated GameObject. Press the button with the red circle to enable record mode. The Animation window time line is **tinted red** when in record mode.

Preview Mode:

The Animation Window in Preview mode

In preview mode, modifying your animated GameObject *does not* automatically create keyframes. You must manually create keyframes (see below) each time you modify your GameObject to a desired new state (for example, moving or rotating it). Press the **Preview** button to enable preview mode. The Animation window time line is **tinted blue** when in preview mode.

*Note: In record mode, the **Preview** button is also active, because you are previewing the existing animation and recording new keyframes at the same time.*

Recording keyframes

To begin recording keyframes for the selected GameObject, click on the **Animation Record button**. This enters **Animation Record Mode**, where changes to the GameObject are recorded into the **Animation Clip**.

Record button

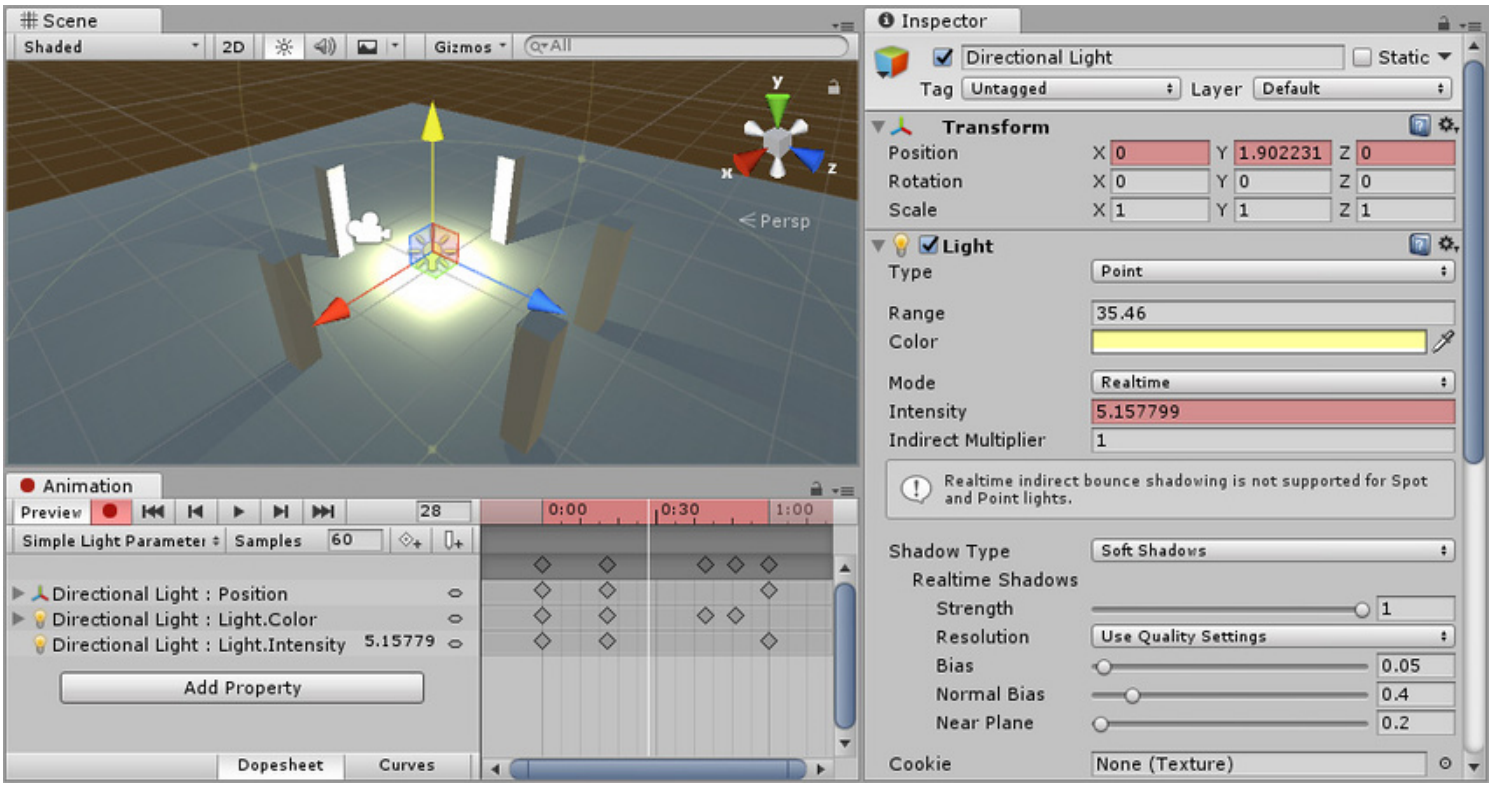
Once in **Record mode** you can create keyframes by setting the white Playback head to the desired time in the Animation time line, and then modify your GameObject to the state you want it to be at that point in time.

The changes you make to the GameObject are recorded as keyframes at the current time shown by the white line (the playback head) in the Animation Window.

Any change to an animatable property (such as its position or rotation) will cause a keyframe for that property to appear in the Animation window.

Clicking or dragging in the time line bar moves the playback head and shows the state of the animation at the playback head's current time.

In the screenshot below you can see the Animation window in record mode. The time line bar is tinted red, indicating record mode, and the animated properties show up with a red background in the inspector.



Current frame

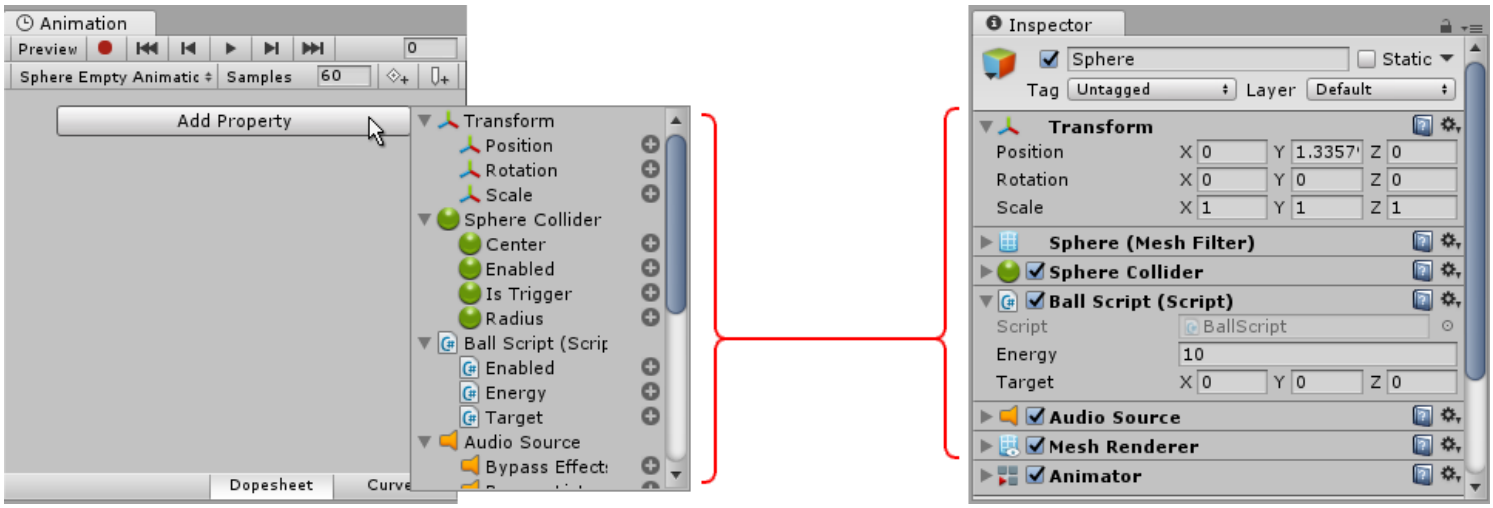
You can stop the **Record Mode** at any time by clicking the **Record button** again. When you stop Record mode, the Animation window switches to **Preview mode**, so that you can still see the GameObject in its current position according to the animation time line.

You can animate any property of the GameObject by manipulating it while in Animation Record Mode. Moving, Rotating or Scaling the GameObject adds corresponding keyframes for those properties in the animation clip. Adjusting values directly in the GameObject's inspector also adds keyframes while in Record mode. This applies to any animatable property in the inspector, including numeric values, checkboxes, colours, and most other values.

Any properties of the GameObject that are currently animated are shown listed in the left-hand side of the Animation Window. Properties which are not animated are not shown in this window. Any new properties that you animate, including properties on child objects, are added to the property list area as soon as you start animating them.

Transform properties are special in that the **.x**, **.y**, and **.z** properties are linked, so curves for all three are added at the same time.

You can also add animatable properties to the current GameObject (and its children) by clicking the **Add Property** button. Clicking this button shows a pop up list of the GameObject's animatable properties. These correspond with the properties you can see listed in the inspector.



The animatable properties of a GameObject are revealed when you click the **Add Property** button

When in **Preview mode** or **Record mode**, the white vertical line shows which frame of the **Animation Clip** is currently previewed. The **Inspector** and **Scene View** shows the GameObject at that frame of the Animation Clip. The values of the animated properties at that frame are also shown in a column to the right of the property names.

In Animation Mode a white vertical line shows the currently previewed frame.

Time line

You can click anywhere on the **Animation window time line** to move the playback head to that frame, and preview or modify that frame in the Animation Clip. The numbers in the **time line** are shown as seconds and frames, so 1:30 means 1 second and 30 frames.

*Note, this **time line** bar in the Animation window shares the same name, but is separate from the **Timeline window***

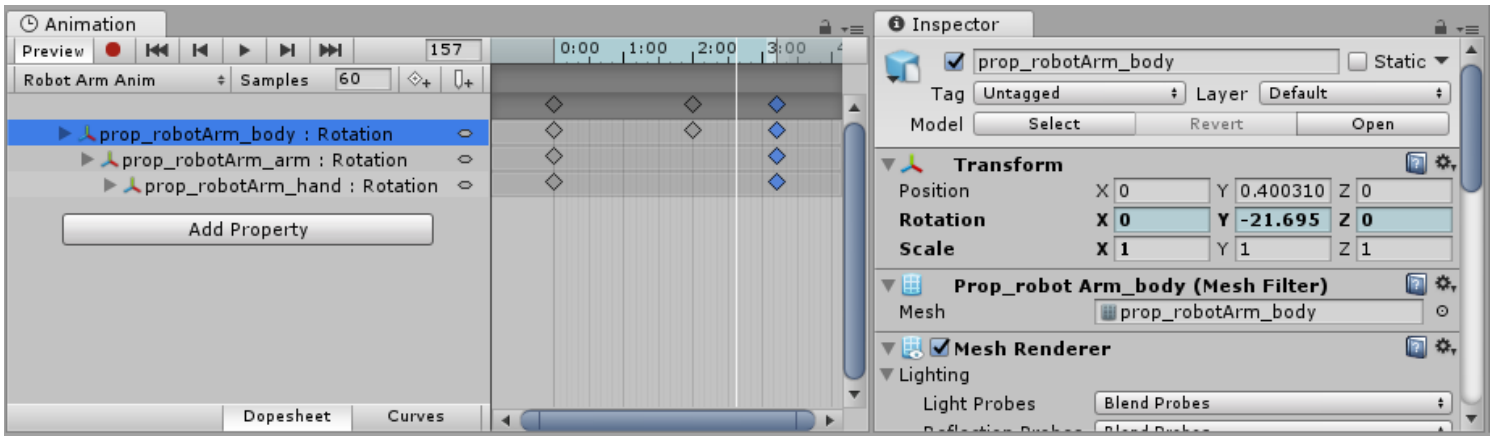
Time Line

Note: The time line appears **tinted blue** when in *Preview* mode, or **tinted red** when in *Record* mode.

Creating keyframes in preview mode

As well as using **Record mode** to automatially create keyframes when you modify an GameObject, you can create keyframes in **Preview mode** by modifying a property on the GameObject, then explicitly choosing to create a keyframe for that property.

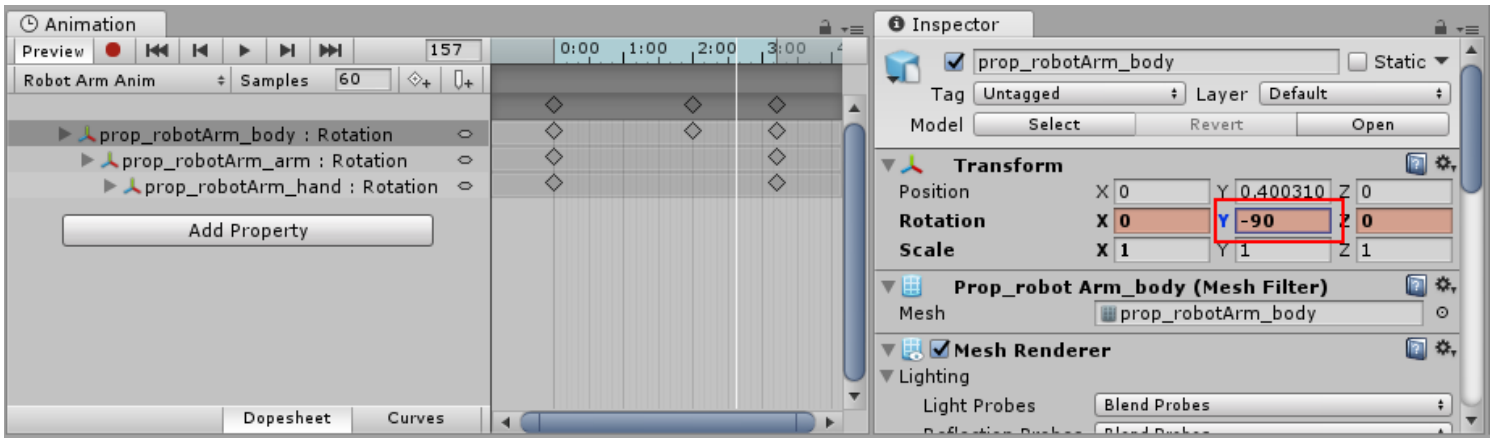
In preview mode, animated properties are tinted blue in the Inspector window. When you see this blue tint, it means these values are being driven by the keyframes of the animation clip currently being previewed in the animation window.



In preview mode, animated fields are tinted blue in the inspector

If you modify any of these blue-tinted properties while previewing (such as rotating a GameObject that has its rotation property animated, as in the above screenshot) the GameObject is now in a **modified** animation state. This is indicated by a change of color in the tint of the inspector field to a pink color. Because you are not in record mode, your modification is not yet saved as a keyframe.

For example, in the screenshot below, the rotation property has been modified to have a Y value of –90. This modification has not yet been saved as a keyframe in the animation clip.



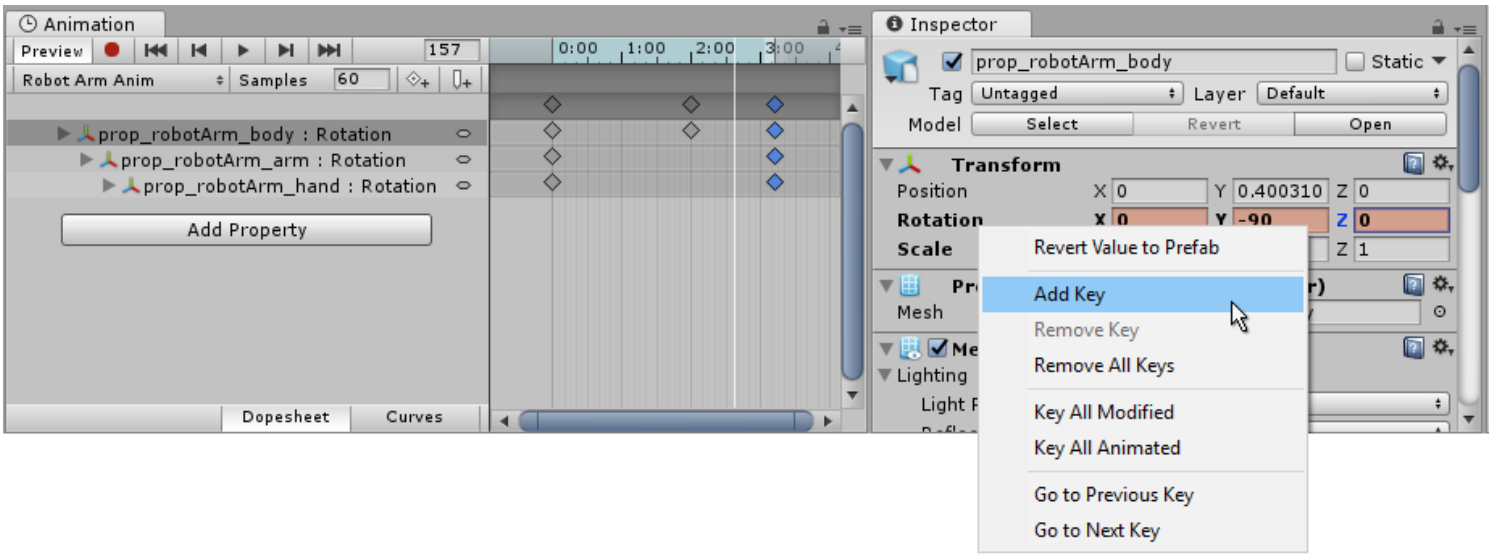
A modified animated property in preview mode. This change has not yet been saved as a keyframe

In this modified state, you must manually create a keyframe to “save” this modification. If you move the playback head, or switch your selection away from the animated GameObject, you will lose the modification.

Manually creating keyframes

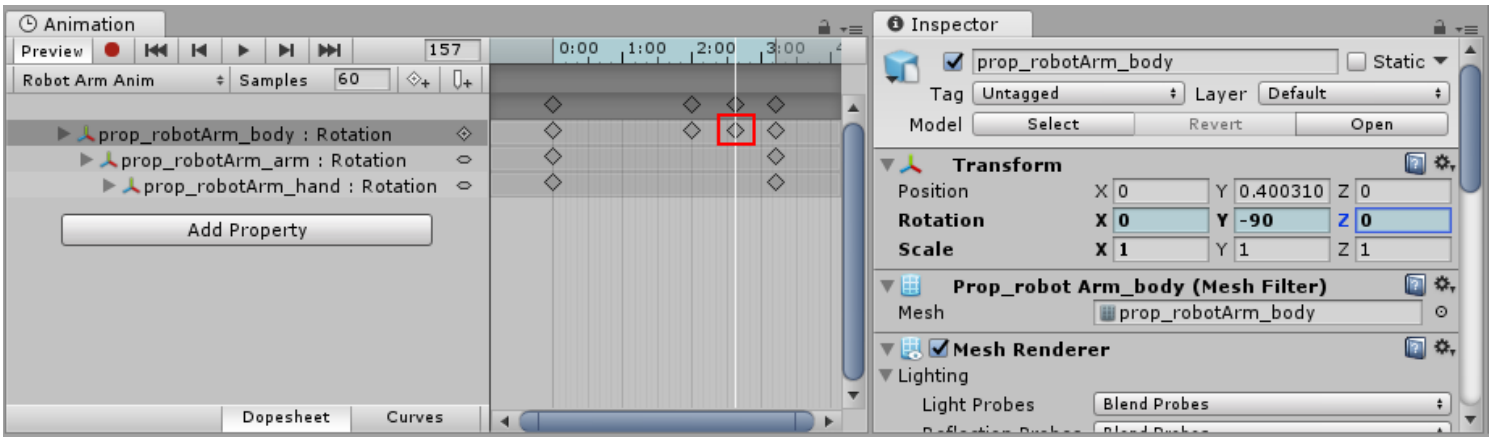
There are three different ways to manually create a keyframe when you have modified a GameObject in preview mode.

You can add a keyframe by right-clicking the **property label** of the property you have modified, which allows you to either add a keyframe for just that property, or for all animated properties:



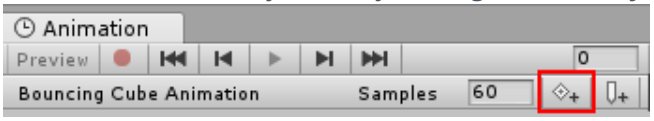
The proprty label context menu

When you have added a keyframe, the new keyframe will be visible in the **animator window** as a diamond symbol (called out in red in the screenshot below), and the property field will return to a blue tint, indicating that your modification is saved as a keyframe, and that you are now previewing a value that is driven by the animation keyframes.



With the new keyframe added (marked in red), the values in the inspector return to a blue tint.

You can also add a keyframe by clicking the **Add Keyframe** button in the Animation window:



Or, you can add a keyframe (or keyframes) by using the hotkeys K or Shift-K as described below:

Hotkeys

K - *Key all animated.* Adds an keyframe for all animated properties at the current position of the playback head in the animation window.

Shift-K - *Key all modified.* Adds an keyframe for only those animated properties which have been modified at the current position of the playback head in the animation window.

2017–09–05 Page amended with limited editorial review

Preview mode added in Unity 2017.1

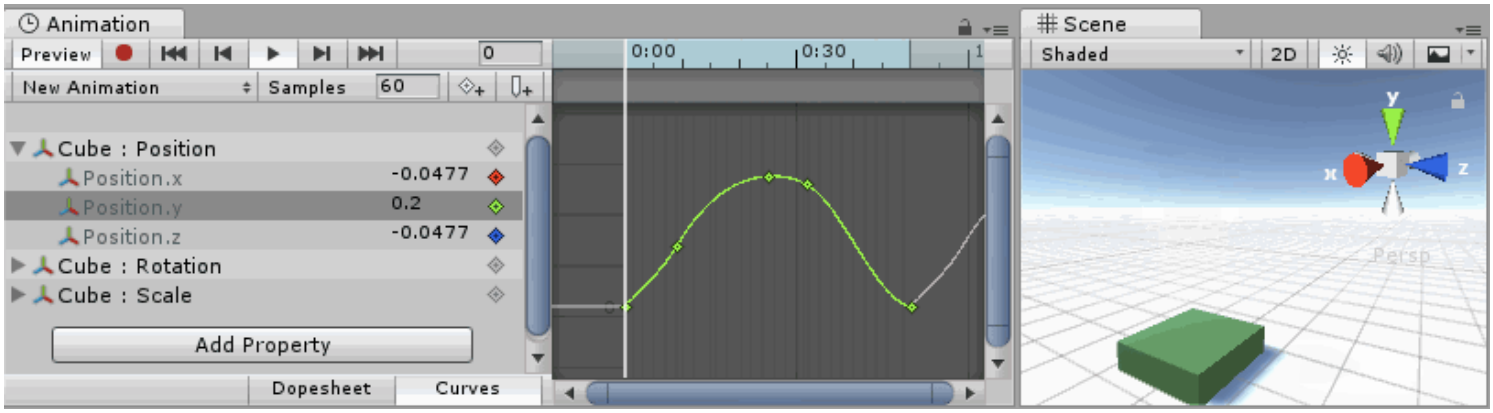
Using Animation Curves

Leave feedback

The Property List

In an **Animation Clip**, any **animatable property** can have an **Animation Curve**, which means that the Animation Clip controls how that property changes over time. In the property list area of the **Animation View** (on the left), all the currently animated properties are listed. With the Animation View in Dope Sheet mode, the animated values for each property appear only as linear tracks, however in Curves mode you are able to see the the changing values of properties visualised as lines on graph. Whichever mode you use to view, the curves still exist - the Dope Sheet mode just gives you a simplified view of the data showing only when the keyframes occur.

In **Curves** mode, the **Animation Curves** have colored curve indicators, each colour representing the values for one of the currently selected properties in the property list. For information on how to add curves to an animation property, see the section on Using the Animation View.

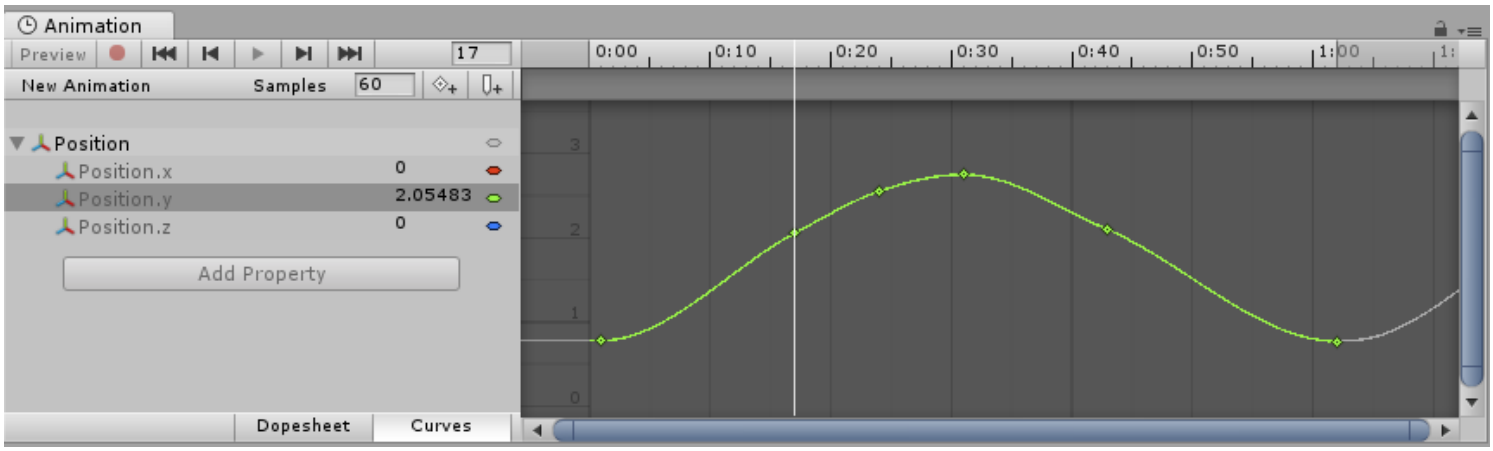


Animation Curves with the color indicators visible. In this example, the green indicator matches the Y position curve of a bouncing cube animation

Understanding Curves, Keys and Keyframes

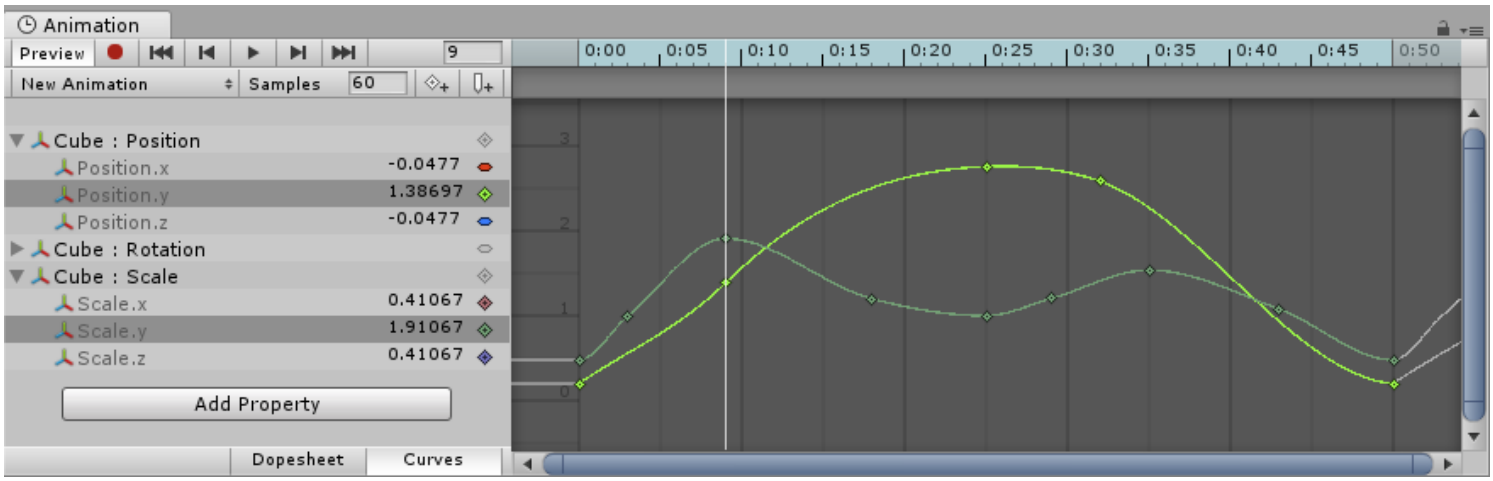
An **Animation Curve** has multiple **keys** which are control points that the curve passes through. These are visualized in the **Curve Editor** as small diamond shapes on the curves. A frame in which one or more of the shown curves have a **key** is called a **keyframe**.

If a property has a **key** in the currently previewed frame, the curve indicator will have a diamond shape, and the property list will also have diamond shapes next to the value.



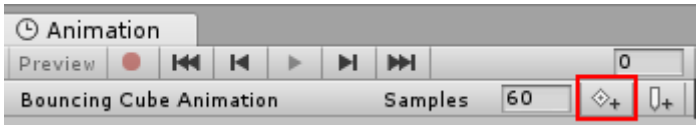
The **Rotation.y** property has a **key** at the currently previewed frame.

The **Curve Editor** will only show curves for the properties that are selected. If multiple properties are selected in the property list, the curves will be shown overlaid together.



When multiple properties are selected, their curves are shown overlaid together in the Curves Editor

Adding and Moving Keyframes



You can add a **keyframe** at the currently previewed frame by clicking the **Keyframe button**.

A **keyframe** can be added at the currently previewed frame by clicking the **Keyframe button**. This will add a keyframe to all currently selected curves. Alternatively you can add a keyframe to a single curve at any given frame by double-clicking the curve where the new **keyframe** should be. It is also possible to add a **keyframe** by right-clicking the **Keyframe Line** and select **Add Keyframe** from the context menu. Once placed, **keyframes** can be dragged around with the mouse. It is also possible to select multiple **keyframes** to drag at once. **Keyframes** can be deleted by selecting them and pressing **Delete**, or by right-clicking on them and selecting **Delete Keyframe** from the context menu.

Supported Animatable Properties

The **Animation View** can be used to animate much more than just the position, rotation, and scale of a **Game Object**. The properties of any **Component** and **Material** can be animated - even the public variables of your own **scripts** components. Making animations with complex visual effects and behaviors is only a matter of adding **Animation Curves** for the relevant properties.

The following types of properties are supported in the animation system:

Float
Color
Vector2
Vector3
Vector4
Quaternion
Boolean

Arrays are not supported and neither are structs or objects other than the ones listed above.

For boolean properties, a value of **0** equals **False** while any other value equals **True**.

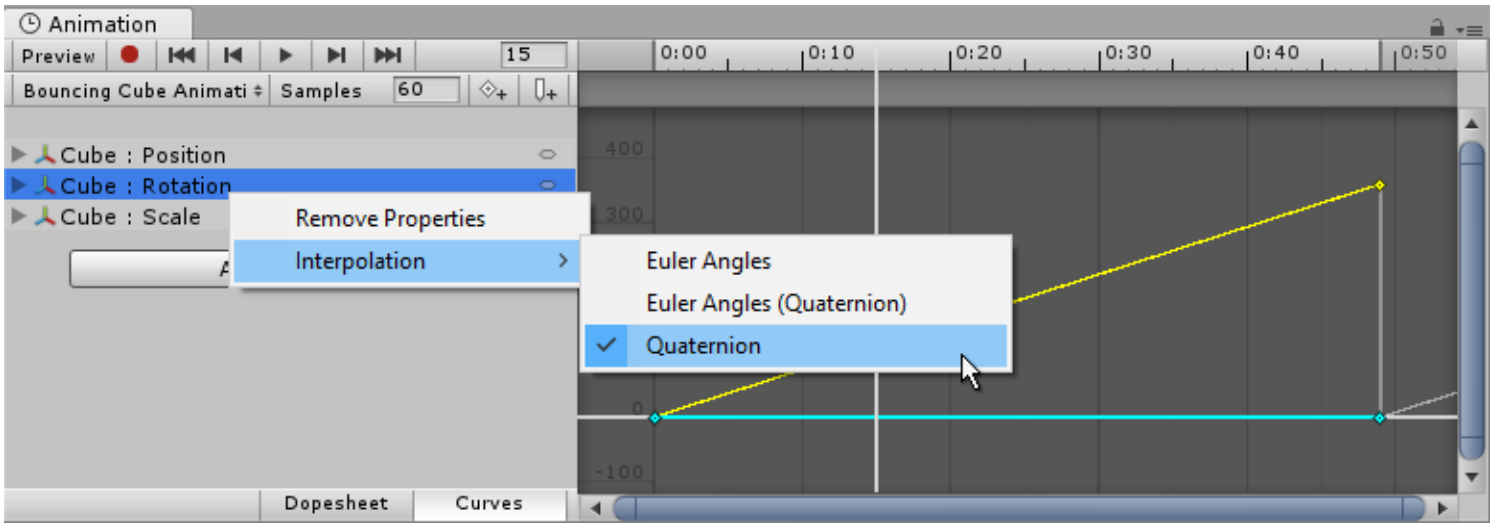
Here are a few examples of the many things the **Animation View** can be used for:

Animate the **Color** and **Intensity** of a **Light** to make it blink, flicker, or pulsate.
Animate the **Pitch** and **Volume** of a looping **Audio Source** to bring life to blowing wind, running engines, or flowing water while keeping the sizes of the sound **assets** to a minimum.
Animate the **Texture Offset** of a **Material** to simulate moving belts or tracks, flowing water, or special effects.
Animate the **Emit** state and **Velocities** of multiple **Ellipsoid Particle Emitters** to create spectacular fireworks or fountain displays.
Animate variables of your own script components to make things behave differently over time.
When using **Animation Curves** to control game logic, please be aware of the way animations are played back and sampled in Unity.

Rotation Interpolation Types

In Unity rotations are internally represented as **Quaternions**. Quaternions consist of **.x**, **.y**, **.z**, and **.w** values that should generally not be modified manually except by people who know exactly what they're doing. Instead, rotations are typically manipulated using **Euler Angles** which have **.x**, **.y**, and **.z** values representing the rotations around those three respective axes.

When interpolating between two rotations, the **interpolation** can either be performed on the **Quaternion** values or on the **Euler Angles** values. The **Animation View** lets you choose which form of **interpolation** to use when animating **Transform** rotations. However, the rotations are always shown in the form of **Euler Angles** values no matter which **interpolation** form is used.



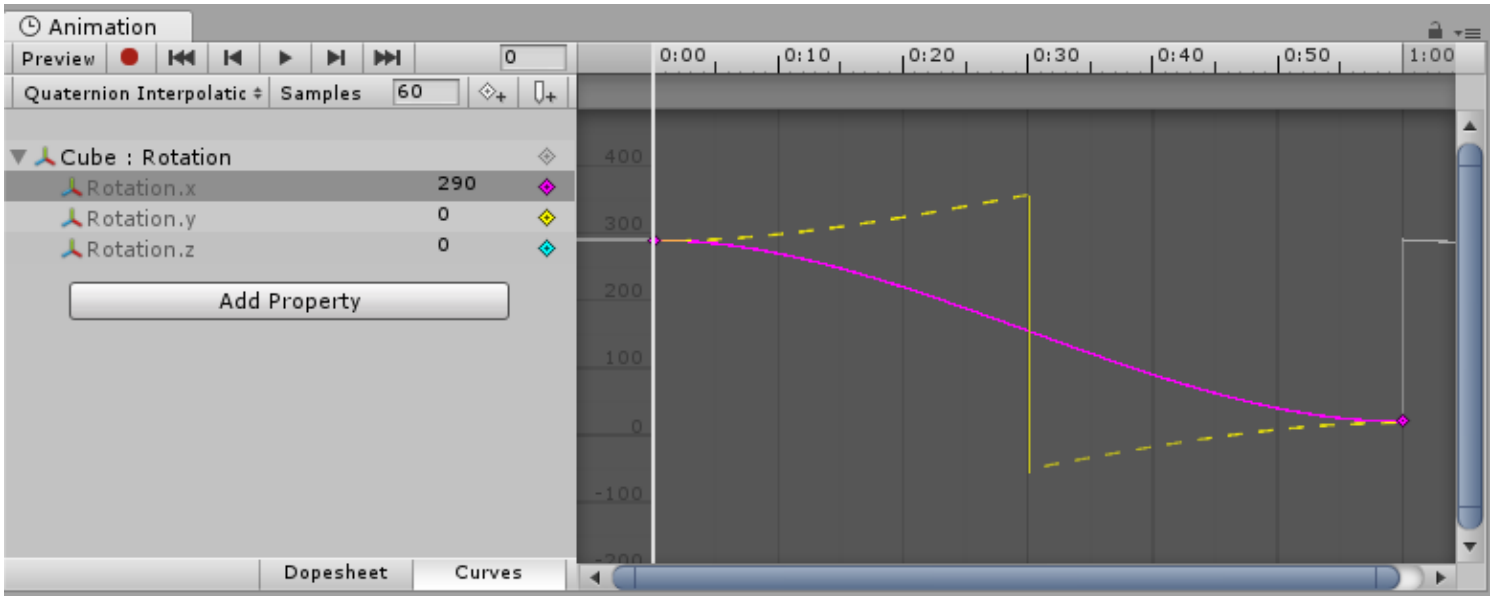
Transform rotations can use **Euler Angles** interpolation or **Quaternion** interpolation.

Quaternion Interpolation

Quaternion interpolation always generates smooth changes in rotation along the shortest path between two rotations. This avoids rotation interpolation artifacts such as Gimbal Lock. However, Quaternion interpolation cannot represent rotations larger than 180 degrees, due to its behaviour of always finding the shortest path. (You

can picture this by picking two points on the surface of a sphere - the shortest line between them will never be more than half-way around the sphere).

If you use Quaternion interpolation and set the numerical rotation values further than 180 degrees apart, the curve drawn in the animation window will still appear to cover more than a 180 degree range, however the actual rotation of the object will take the shortest path.



Placing two keys 270 degrees apart when using Quaternion interpolation will cause the interpolated value to go the other way around, which is only 90 degrees. The magenta curve is what is actually shown in the animation window. The true interpolation of the object is represented by the yellow dotted line in this screenshot, but does not actually appear in the editor.

When using Quaternion interpolation for rotation, changing the keys or tangents of either the x, y or z curve may also change the values of the other two curves, since all three curves are created from the internal Quaternion representation. When using Quaternion interpolation, keys are always linked, so that creating a key at a specific time for one of the three curves (x, y or z) will also create a key at that time for the other two curves.

Euler Angles Interpolation

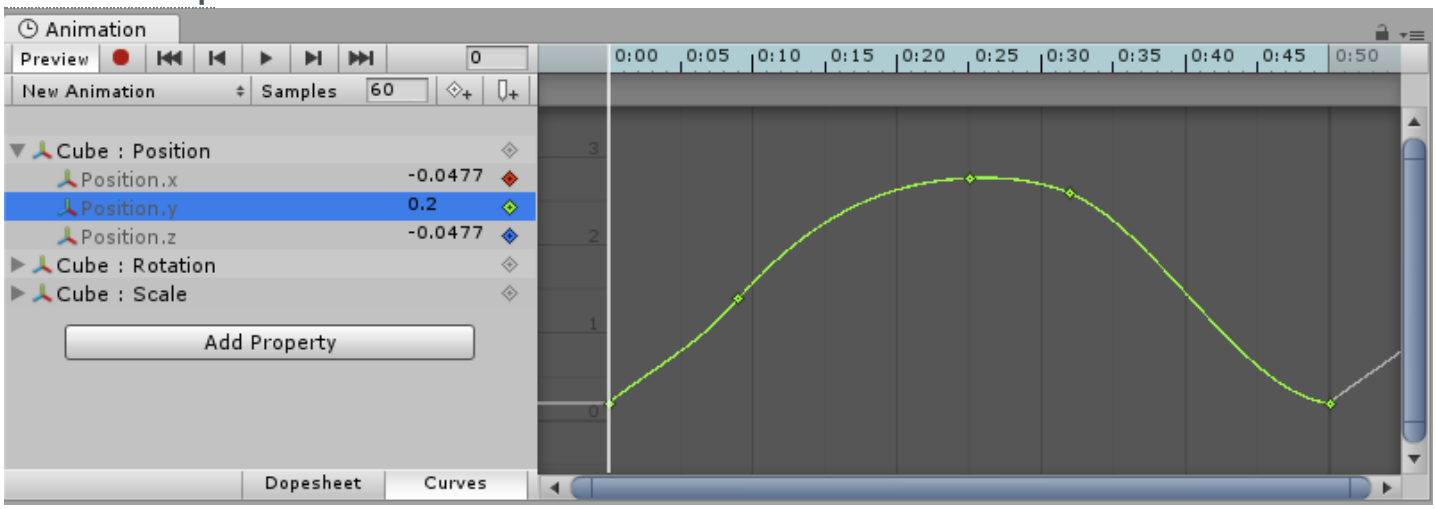
Euler Angles interpolation is what most people are used to working with. Euler Angles can represent arbitrary large rotations and the **.x**, **.y**, and **.z** curves are independent from each other. Euler Angles interpolation can be subject to artifacts such as Gimbal Lock when rotating around multiple axes at the same time, but are intuitive to work with for simple rotations around one axis at a time. When Euler Angles interpolation is used, Unity internally bakes the curves into the Quaternion representation used internally. This is similar to what happens when importing animation into Unity from external programs. Note that this curve baking may add extra keys in the process and that tangents with the **Constant** tangent type may not be completely precise at a sub-frame level.

Editing Curves

Leave feedback

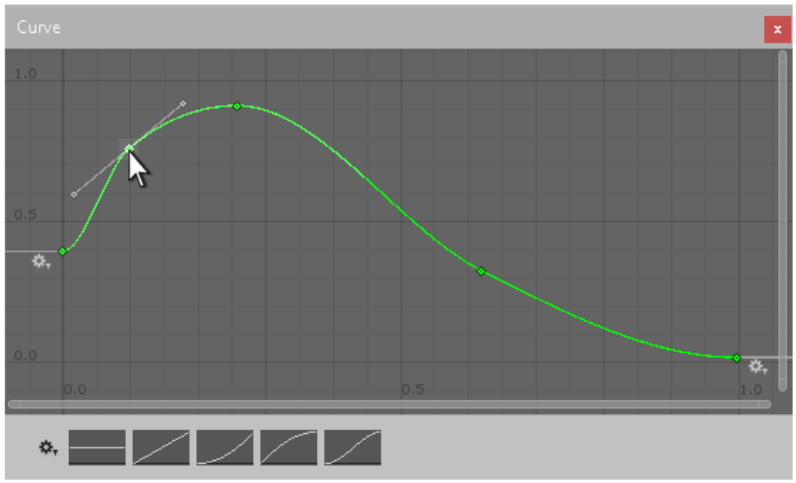
There are several different features and windows in the Unity Editor which use **Curves** to display and edit data. The methods you can use to view and manipulate curves is largely the same across all these areas, although there are some exceptions.

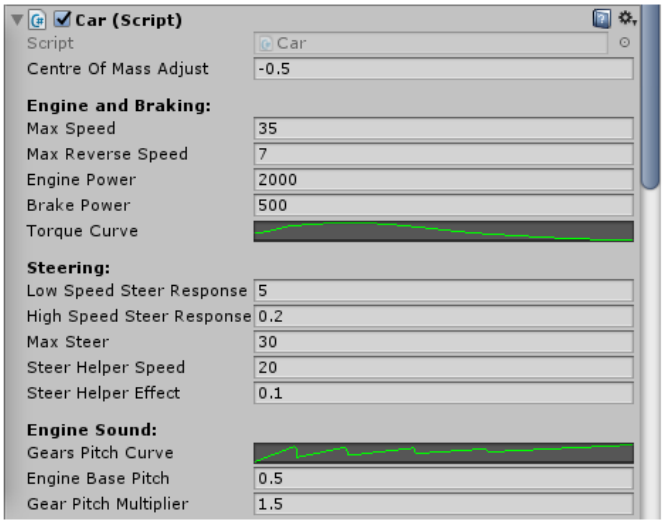
The Animation Window uses curves to display and edit the values of animated properties over time in an **Animation Clip**.



The Animation Window.

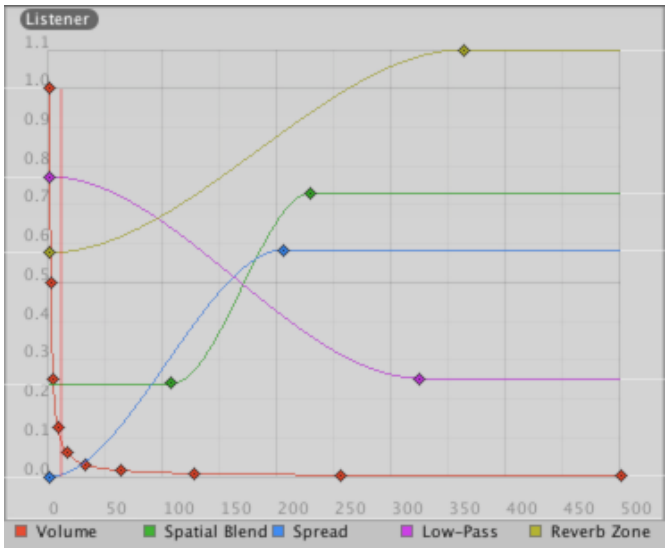
Script **components** can have member variables of type Animation Curve that can be used for all kinds of things. Clicking on those in the **Inspector** will open up the **Curve Editor**.





The Curve Editor.

The Audio Source component uses curves to control roll-off and other properties as a function of distance to the Audio Source.



Distance function curves in the AudioSource component in the Inspector.

While these controls have subtle differences, the **curves** can be edited in exactly the same way in all of them. This page explains how to navigate and edit curves in those controls.

Adding and Moving Keys on a Curve

A key can be added to a curve by double-clicking on the curve at the point where the **key** should be placed. It is also possible to add a **key** by right-clicking on a curve and select **Add Key** from the context menu.

Once placed, **keys** can be dragged around with the mouse:

Click on a **key** to select it. Drag the selected **key** with the mouse.
To snap the **key** to the grid while dragging it around, hold down **Command** on Mac / **Control** on Windows while dragging.

It is also possible to select multiple **keys** at once:

To select multiple **keys** at once, hold down **Shift** while clicking the keys.
To deselect a selected **key**, click on it again while holding down **Shift**.
To select all **keys** within a rectangular area, click on an empty spot and drag to form the rectangle selection.
The rectangle selection can also be added to existing selected keys by holding down **Shift**.

Keys can be deleted by selecting them and pressing **Delete**, or by right-clicking on them and selecting **Delete Key** from the context menu.

Editing Keys

Direct editing of key values in curve editors is a new feature in Unity 5.1. Use **Enter/Return** or context menu to start editing selected keys, **Tab** to switch between fields, **Enter/Return** to commit, and **Escape** to cancel editing.

Navigating the Curve View

When working with the **Animation View** you can easily zoom in on details of the curves you want to work with or zoom out to get the full picture.

You can always press **F** to frame-select the shown curves or selected keys in their entirely.

Zooming

You can **zoom** the Curve View using the scroll-wheel of your mouse, the zoom functionality of your trackpad, or by holding **Alt** while right-dragging with your mouse.

You can zoom on only the horizontal or vertical axis:

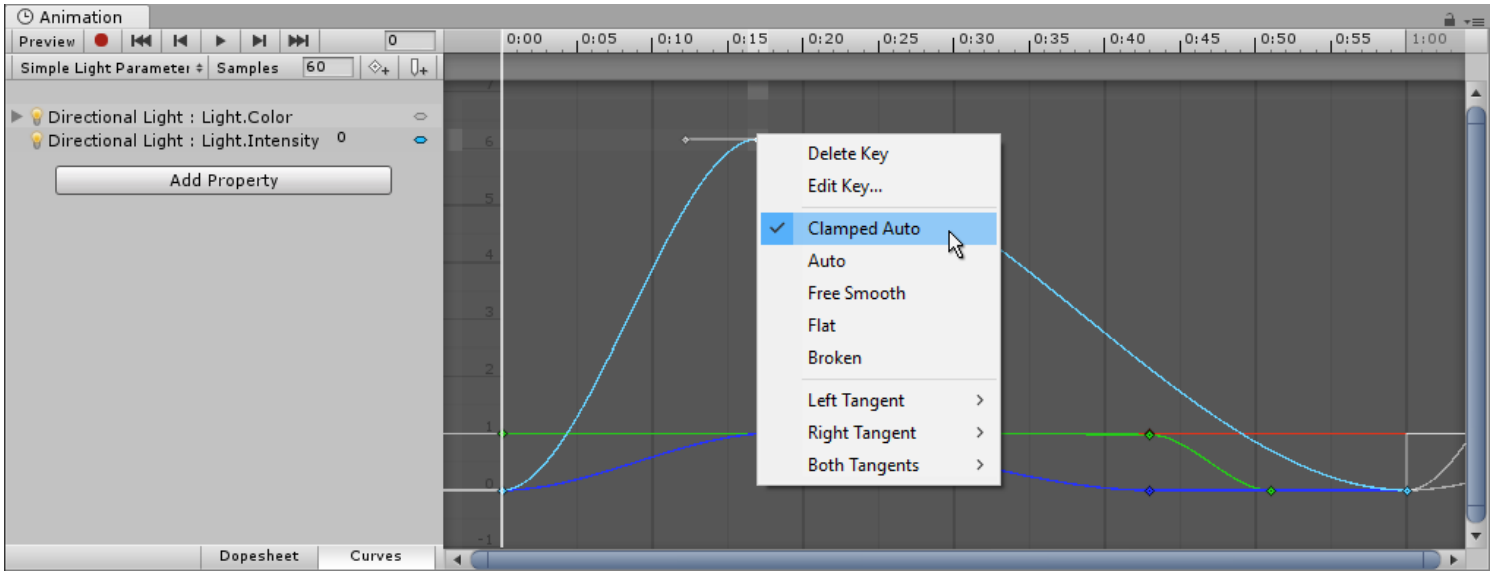
zoom while holding down **Command** on Mac / **Control** on Windows to zoom horizontally.
zoom while holding down **Shift** to zoom vertically.
Furthermore, you can drag the end caps of the scrollbars to shrink or expand the area shown in the Curve View.

Panning

You can **pan** the Curve View by middle-dragging with your mouse or by holding **Alt** while left-dragging with your mouse.

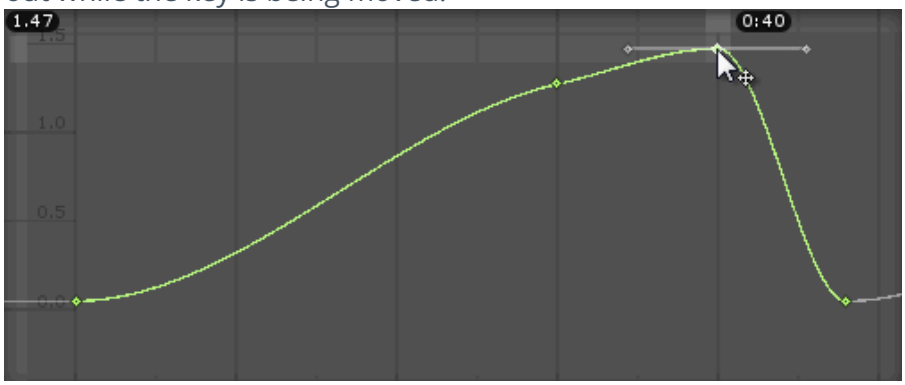
Editing Tangents

A key has two **tangents** - one on the left for the incoming slope and one on the right for the outgoing slope. The tangents control the shape of the curve between the keys. You can select from a number of different tangent types to control how your curve leaves one key and arrives at the next key. Right-click a key to select the tangent type for that key.



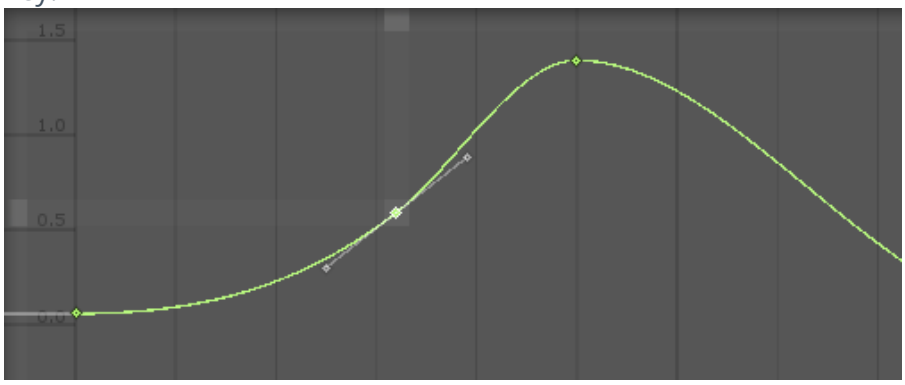
For animated values to change smoothly when passing a key, the left and right tangent must be co-linear. The following tangent types ensure smoothness:

Clamped Auto: This is the default **tangent mode**. The tangents are automatically set to make the curve pass smoothly through the key. When editing the key's position or time, the tangent adjusts to prevent the curve from "overshooting" the target value. If you manually adjust the tangents of a key in Clamped Auto mode, it is switched to **Free Smooth** mode. In the example below, the tangent automatically goes into a slope and levels out while the key is being moved:

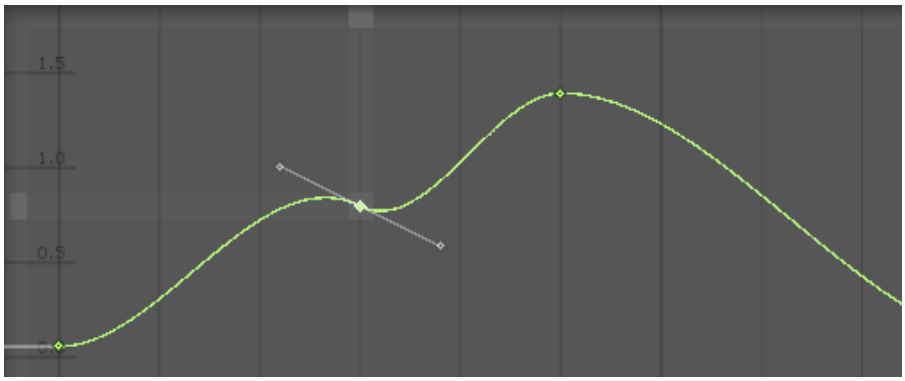


Auto: This is a Legacy tangent mode, and remains an option to be backward compatible with older projects. Unless you have a specific reason to use this mode, use the default **Clamped Auto**. When a key is set to this mode, the tangents are automatically set to make the curve pass smoothly through the key. However, there are two differences compared with **Clamped Auto** mode:
The tangents do not adjust automatically when the key's position or time is edited; they only adjust when initially setting the key to this mode.

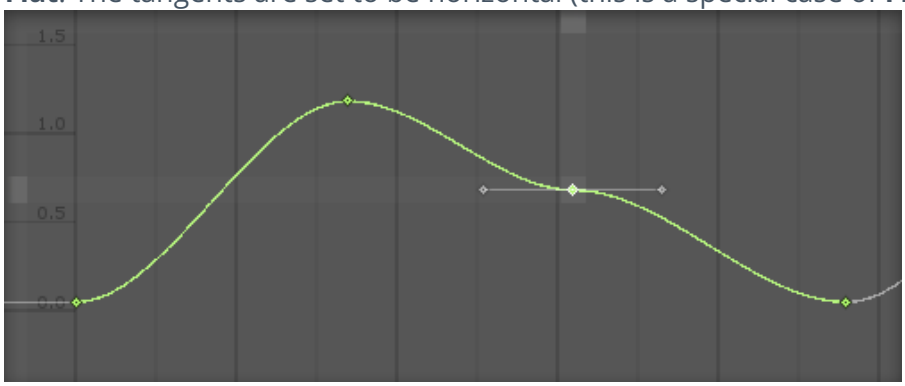
When Unity calculates the tangent, it does not take into account avoiding “overshoot” of the target value of the key.



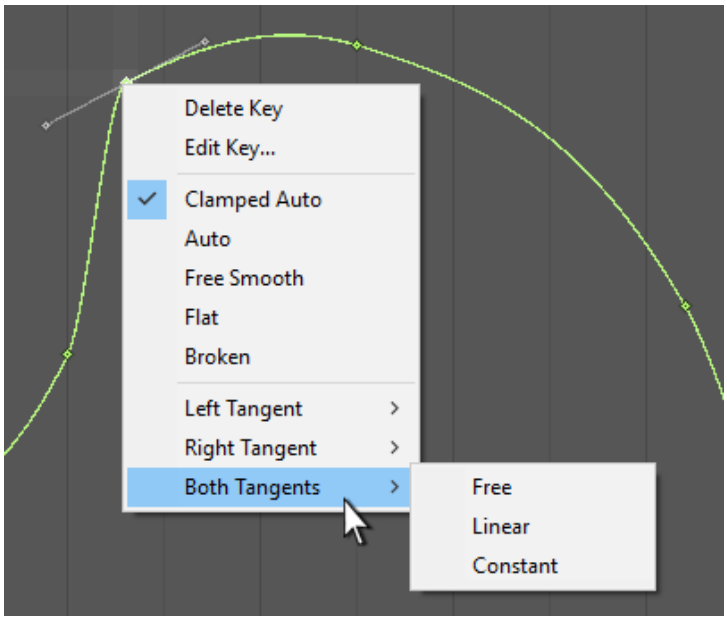
Free Smooth: Drag the tangent handles to freely set the tangents. They are locked to be co-linear to ensure smoothness.



Flat: The tangents are set to be horizontal (this is a special case of **Free Smooth**).

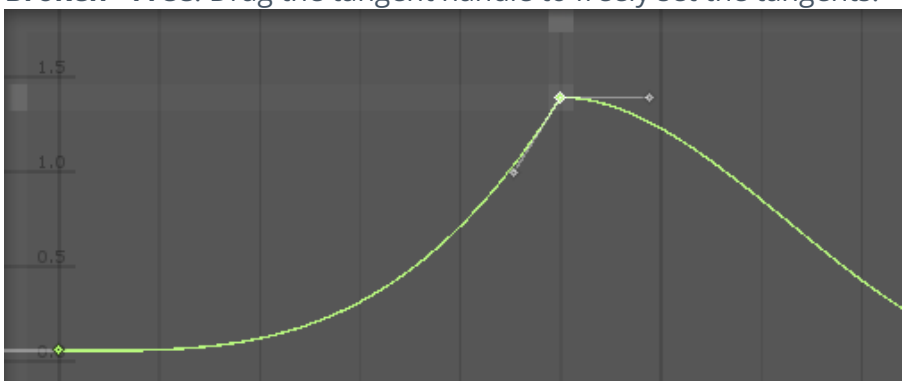


Sometimes you might not want the curve to be smooth when passing through a key. To create sharp changes in the curve, choose one of the **Broken** tangent modes.

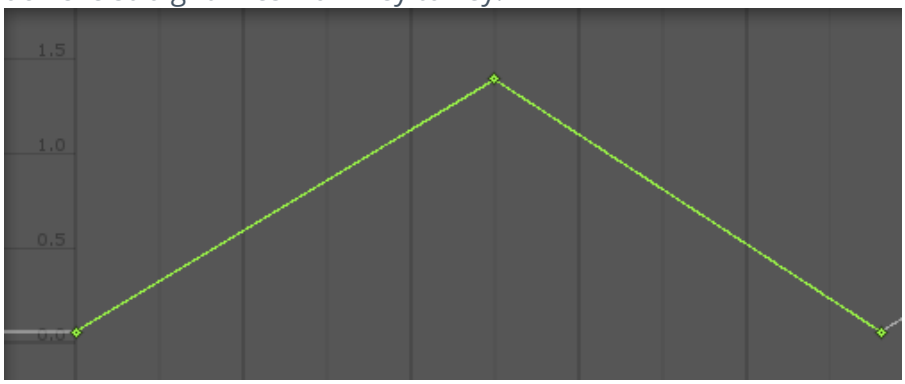


When using broken tangents, the left and right tangent can be set individually. Each of the left and right tangents can be set to one of the following types:

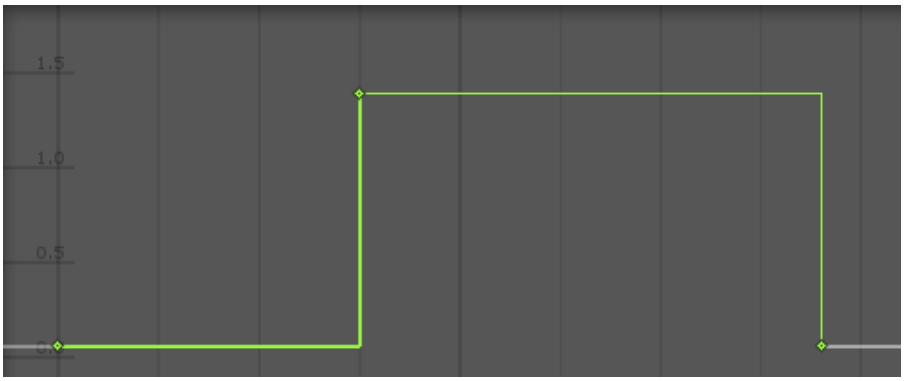
Broken - Free: Drag the tangent handle to freely set the tangents.



Broken - Linear: The tangent points towards the neighboring key. To make a linear curve segment, set the tangents at both ends to be **Linear**. In the example below, all three keys have been set to **Broken - Linear**, to achieve straight lines from key to key.



Broken - Constant: The curve retains a constant value between two keys. The value of the left key determines the value of the curve segment.


1.5
1.0
0.5
0.0

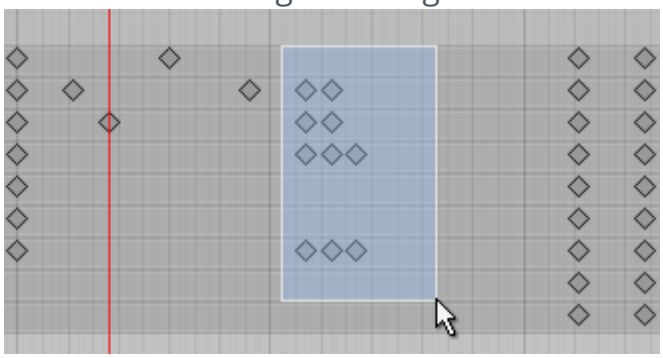
Key manipulation in Dopesheet mode

Leave feedback

Box Selection is used to select multiple keys while viewing the Animation window in **Dopesheet** mode. This allows you to select and manipulate several keys at once.

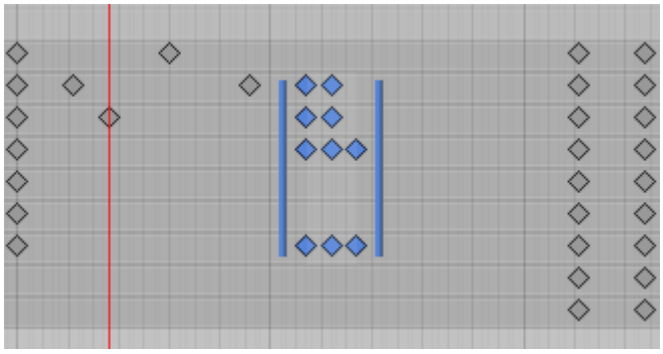
The following actions allow you to select multiple keys:

Hold Shift+click to add individual keys to your selection
Drag a rectangle with the mouse to select a group of keys
Hold Shift and drag a rectangle to add or remove group of keys to the current selection



A rectangle dragged across to select multiple keys in **Dopesheet** mode

As you add keys to the selection, Box Selection handles appear on either side of the selected keys. If you add or remove more keys to the selection, the handles automatically adapt their position and size to enclose all the currently selected keys.



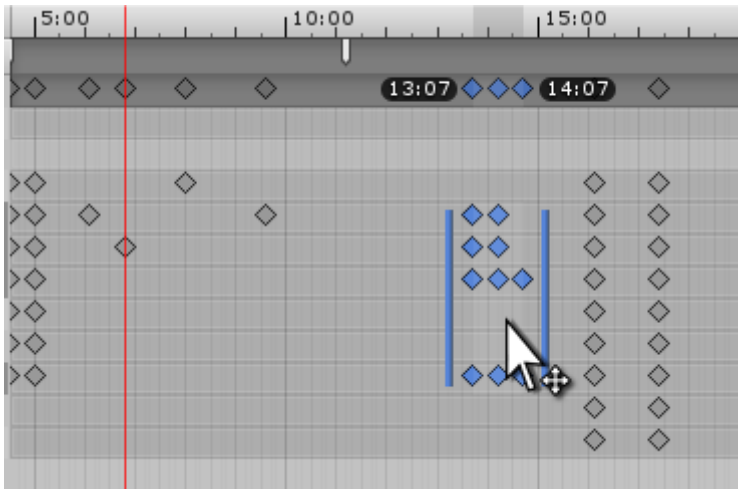
Box Selection handles, displayed to the left and right of the selected keys

Use the Box Selection handles to move, scale and ripple-edit the selected keys (see **Ripple editing**, below).

Moving selected keys

Click anywhere within the Box Selection handles to drag the selected keys and move them. You do not need to click directly on a key to do this; you can drag by clicking the empty space within the Box Selection handles.

While you drag, the time of first and last key is displayed under the **timeline** bar to help you place your keys at the desired position. While dragging a selection of keys to the left, any keys that end up at a negative time (that is, to the left of the 0 marker on the timeline) are deleted when you release the mouse button.

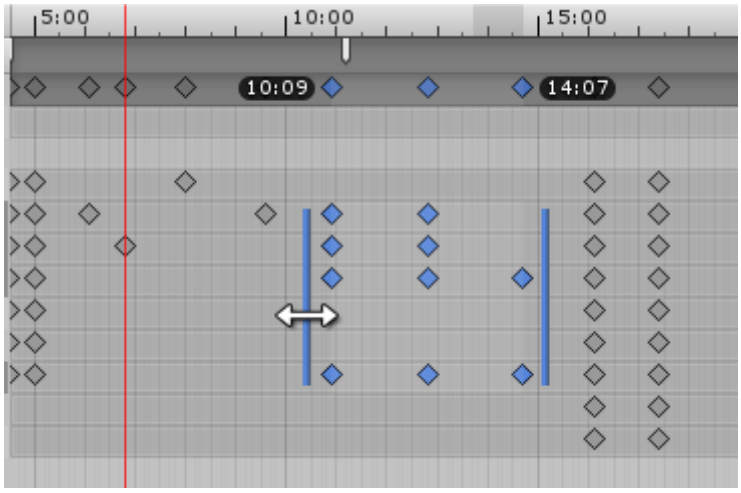


Dragging a selection of keys. Note the start and end times of the selection displayed under the top timeline bar.

Scaling selected keys

When you have multiple keys selected, you can **Scale** the selected keys, either pulling them apart over a longer period of time (making the selected animation slower), or pushing them closer together to occupy a shorter period of time (making the selected animation faster). To scale the selected keys, click and hold either of the blue Box Selection handles at the left and right side of the selected keys, and drag horizontally.

While you scale, the time of first and last key is displayed under the timeline bar to help you scale your keys to the desired position. When scaling a selection of keys down to a smaller amount of time, some keys might end up on the same frame as each other. If this happens, the extra keys that occupy the same frame are discarded when you release the mouse button, and only the last key is kept.



Scaling a selection of keys. Note the start and end times of the selection displayed under the top timeline bar.

Ripple editing

Ripple editing is a method of moving and scaling selected keys. This method also affects non-selected keys on the same timeline as the keys that you are manipulating. The name refers to having the rest of your content automatically move along the timeline to accommodate content you have added, expanded or shrunk. The effects of your edit have a “ripple effect” along the whole timeline.

Press and hold the **R** key while dragging inside the Box Selection to perform a **Ripple Move**. This has the effect of “pushing” any unselected keys, plus the original amount of space between your selection and those keys, to the left or right of your selection when you drag the selected keys along the timeline.

Press and hold the **R** key while dragging a Box Selection handle to perform a **Ripple Scale**. The effect on the rest of the unselected keys in the timeline is exactly the same as with a Ripple Move - they are pushed to the left or right as you scale to the left or right side of your Box Selection.

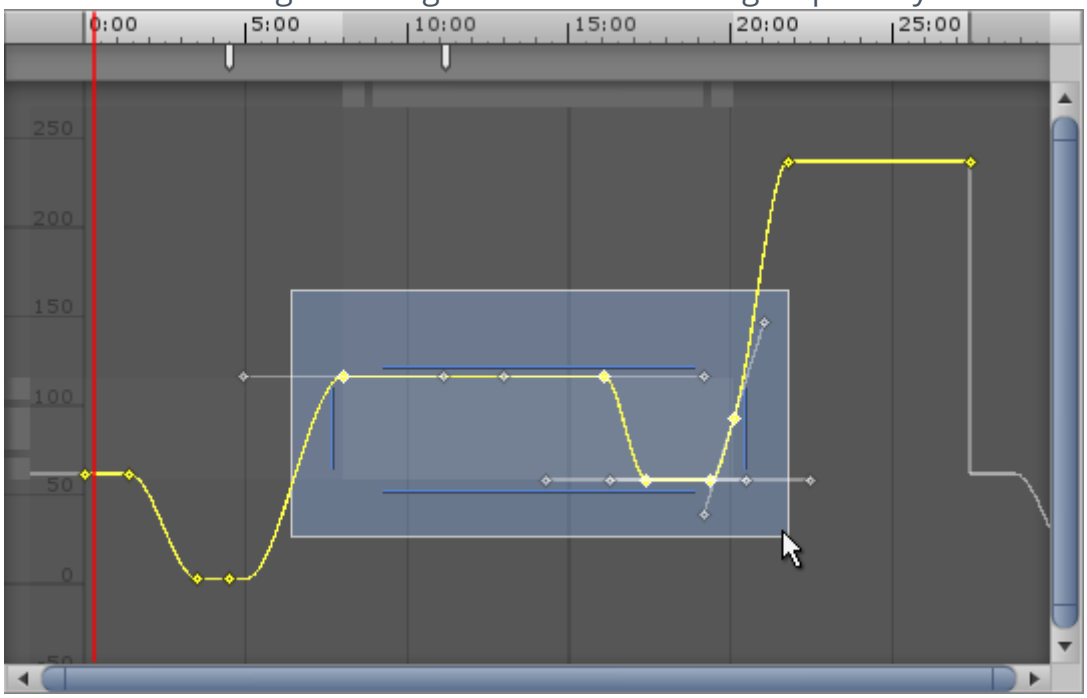
Key manipulation in Curves mode

Leave feedback

Box Selection is used to select multiple keys while viewing the Animation window in **Curves** mode. This allows you to select and manipulate several keys at once.

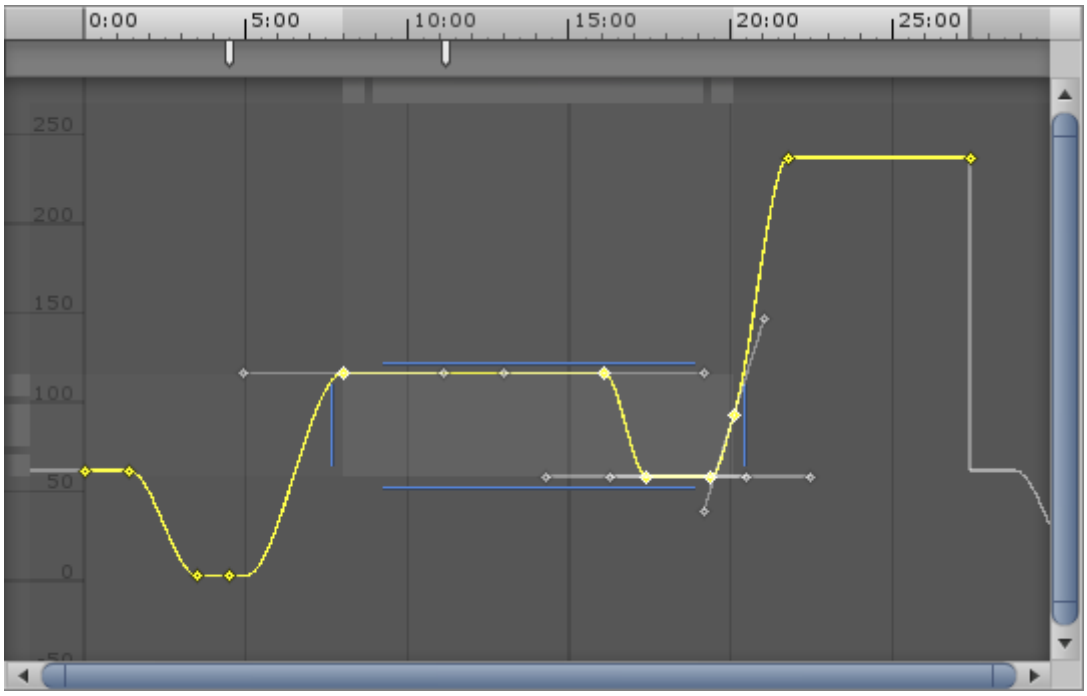
The following actions allow you to select multiple keys:

Hold Shift+click to add individual keys to your selection
Drag a rectangle with the mouse to select a group of keys
Hold Shift and drag a rectangle to add or remove group of keys to the current selection



A rectangle dragged across to select multiple keys in **Curves** mode

As you add keys to the selection, Box Selection handles appear on either side of the selected keys, and at the top and bottom. If you add or remove more keys to the selection, the handles automatically adapt their position and size to enclose all the currently selected keys.

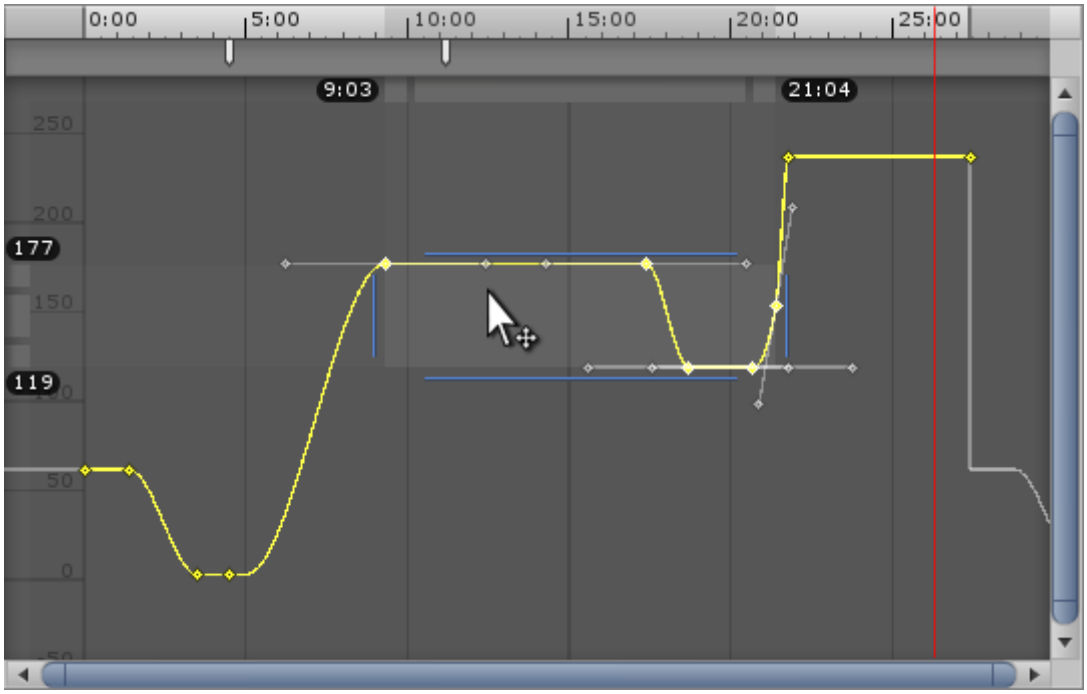


Box Selection handles, displayed on all sides of the selected keys

Moving selected keys

Click anywhere within the Box Selection handles to drag the selected keys and move them. You do not need to click directly on a key to do this; you can drag by clicking the empty space within the Box Selection handles.

While you drag, the time of first and last key is displayed under the **timeline** bar to help you place your keys at the desired position. While dragging a selection of keys to the left, any keys that end up at a negative time (that is, to the left of the 0 marker on the timeline) are deleted when you release the mouse button.



Dragging a selection of keys. Note the start and end times of the selection displayed under the top timeline bar.

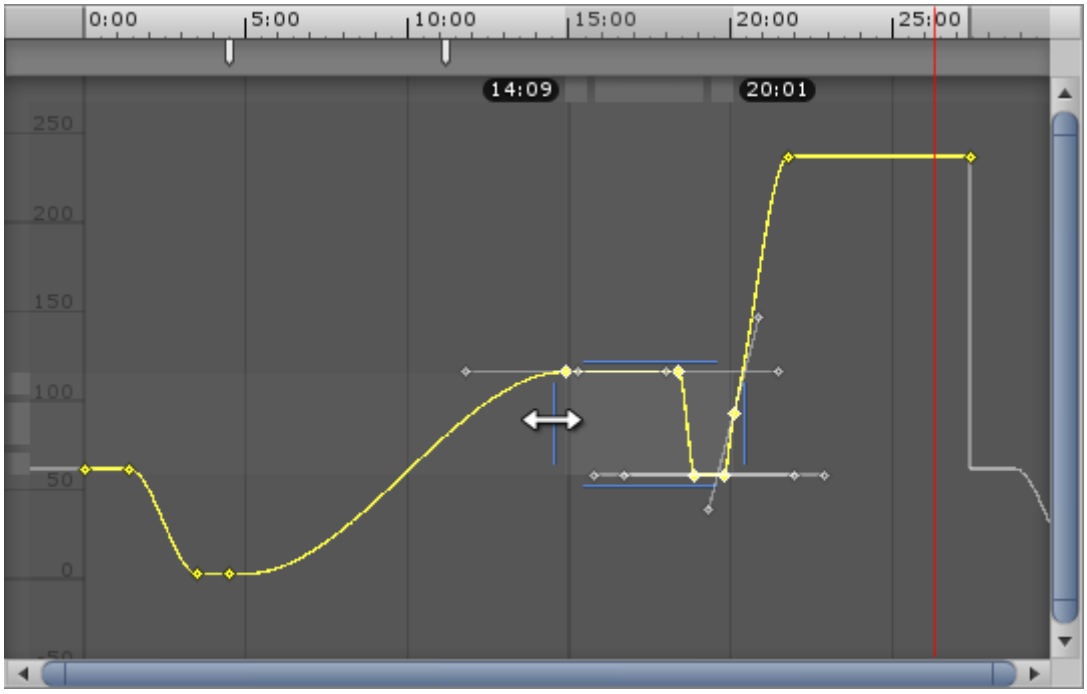
Scaling selected keys

When you have multiple keys selected, you can **Scale** the selected keys. In Curve mode, you can scale horizontally to change the time placement of the keys, or vertically to change the value of the keys.

Horizontally scaling selected keys

Use the Box Selection handles to the left and right of the selected keys to scale the selection horizontally. This changes the time placement of the keys without modifying their values. Pull the handles apart to stretch the keys over a longer period of time (making the selected animation slower), or push them closer together to place the keys over a shorter period of time (making the selected animation faster).

While you scale the selection horizontally, the minimum and maximum time of the keys are shown at the top of the view, to help you set your selection to the desired time.

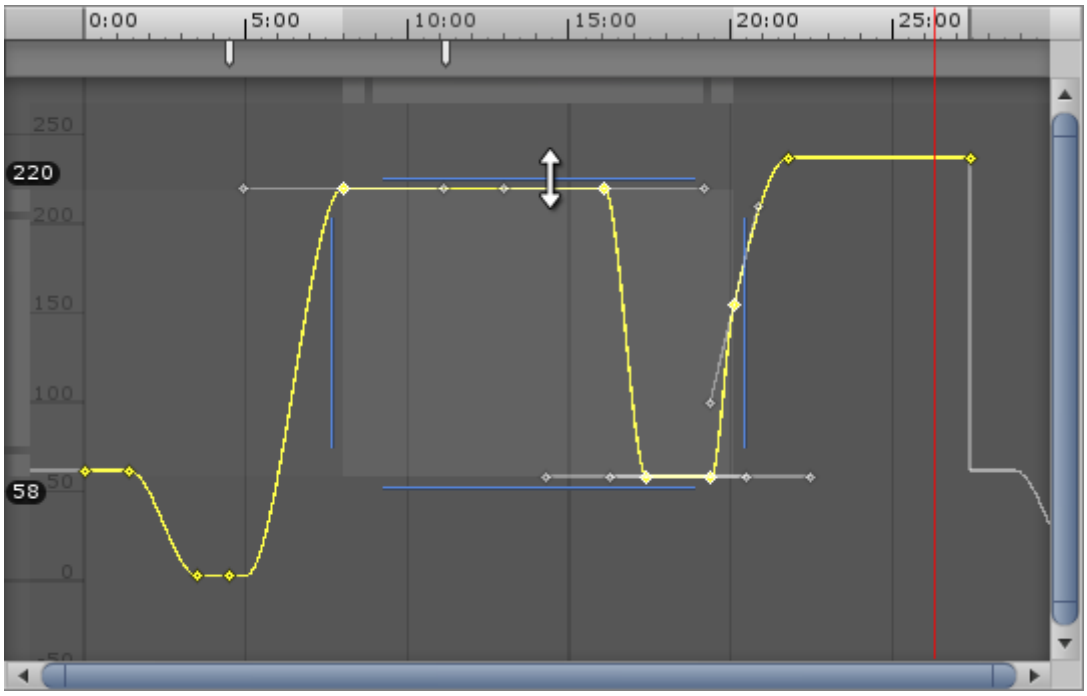


Scaling a selection of keys horizontally. Note the minimum and maximum times of the scaled keys shown at the top of the view.

Vertically scaling selected keys

Use the Box Selection handles to the top and bottom of the selected keys to scale the selection vertically. This changes the value of the keys without modifying their time placement.

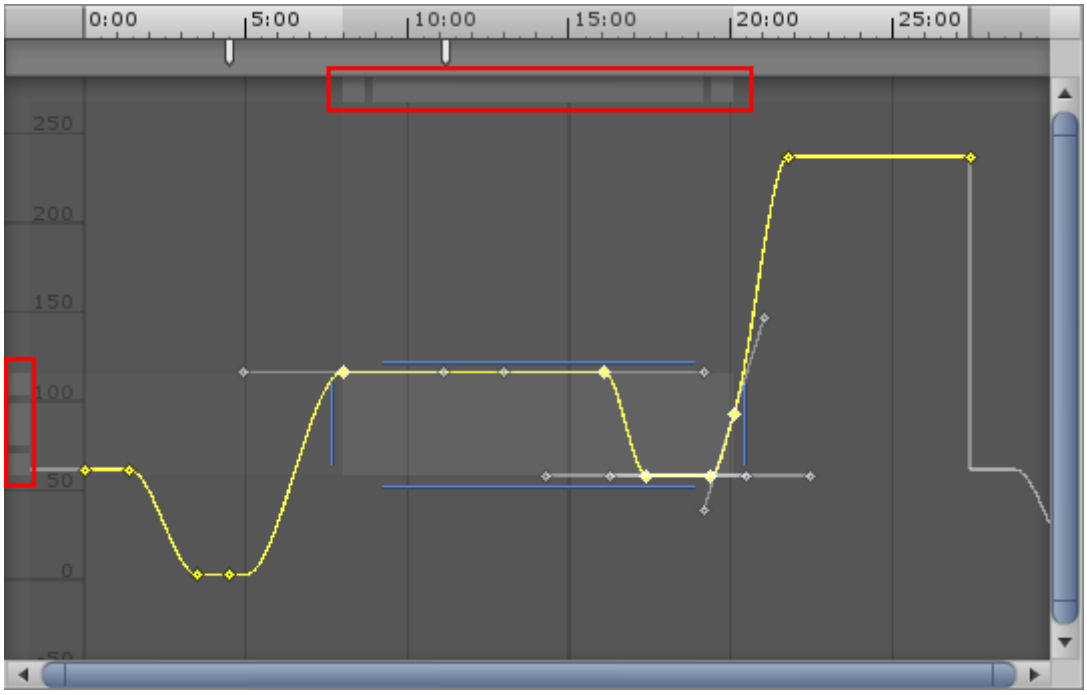
While you scale the selection vertically, the minimum and maximum time of the keys are shown to the left of the view, to help you set your selection to the desired values.



Scaling a selection of keys vertically, modifying their values while preserving their time. Note the new minimum and maximum values of the scaled keys shown to the left if the view.

Manipulation bars

In addition to the Box Selection handles that appear around your selected keys, there are also grey manipulation bars to the top and left of the **Curves** window, which provide additional ways to manipulate the current selection.



The manipulation bars, highlighted in red

The manipulation bar at the top allows you to modify the time placement of the selected keys, while preserving their values. The bar at the side allows you to modify the values of the keys while preserving their time placement.

When you have multiple keys selected, the bars at the top and right display a square at each end. Drag the centre of the bar to move the selected keys (either horizontally or vertically), or drag the squares at the end of each bar to scale the selected keys.

Just like the Box Selection handles, when moving or scaling the selected keys using grey bars, the minimum and maximum values or **keyframe** times are shown. For the time manipulation bar (at the top of the window), the times of the first and last keyframes are displayed. For the value manipulation bar (at the left of the window), the lowest and highest values of the keys are displayed.

Note: the scale boxes at the end of the bars are only visible if you have multiple keys selected, and the view is sufficiently zoomed in so that the bar is long enough to show the scale boxes at each end.

Ripple editing

Ripple editing is a method of moving and scaling selected keys. This method also affects non-selected keys on the same timeline as the keys that you are manipulating. The name refers to having the rest of your content automatically move along the timeline to accommodate content you have added, expanded or shrunk. The effects of your edit have a “ripple effect” along the whole timeline.

Press and hold the **R** key while dragging inside the Box Selection to perform a **Ripple Move**. This has the effect of “pushing” any unselected keys, plus the original amount of space between your selection and those keys, to the left or right of your selection when you drag the selected keys along the timeline.

Press and hold the **R** key while dragging a Box Selection handle to perform a **Ripple Scale**. The effect on the rest of the unselected keys in the timeline is exactly the same as with a Ripple Move - they are pushed to the left or right as you scale to the left or right side of your Box Selection.

Objects with Multiple Moving Parts

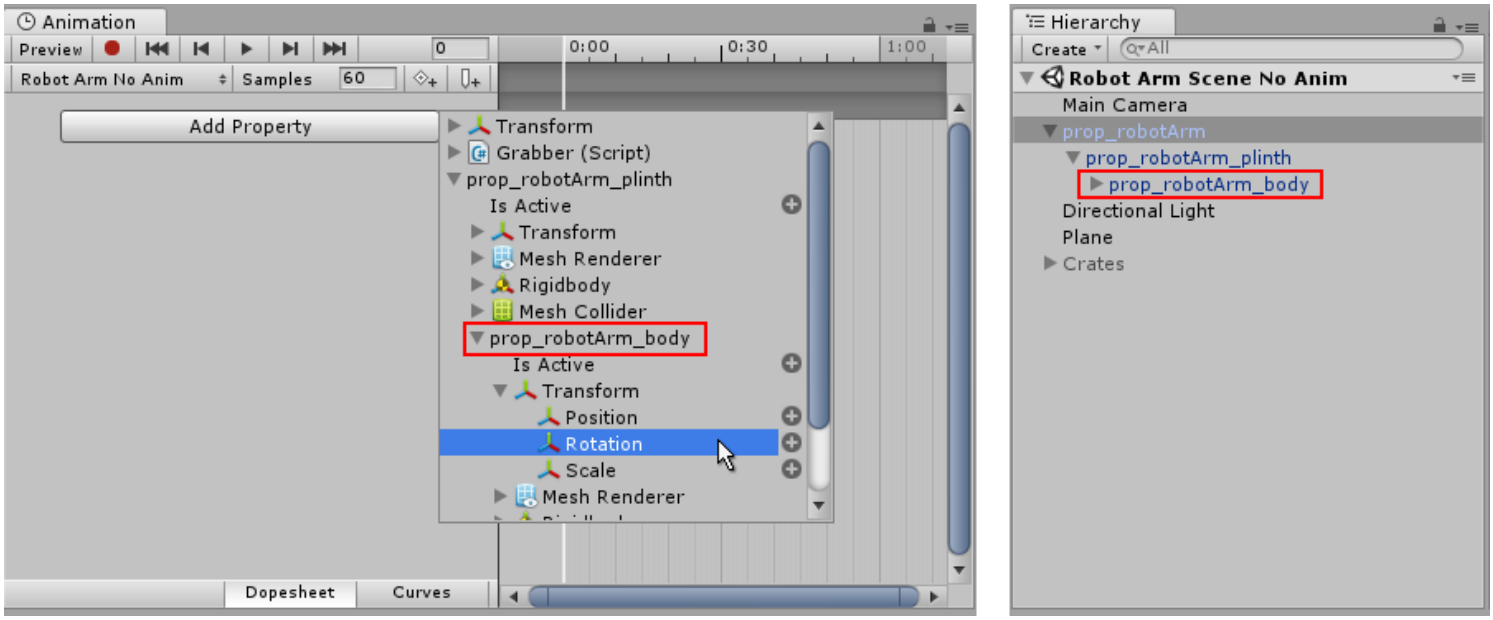
Leave feedback

You may want to animate **Game Objects** that have multiple moving parts, such as a gun turret with a moving barrel, or a character with many body parts. All the parts can be animated by a single Animation **component** on the parent, although it is useful to have additional Animation components on the children in some cases.

Animating Child Game Objects

The Game Object hierarchy is shown in the panel to the left of the **Animation View**.

You can access the children of a Game Object by using the foldout triangle next to the object's name. The properties of child objects can be animated just like those of the parent.



Child **Game Objects** appear in the list of animatable properties when pressing the Add Curve button. They can be expanded to view the animatable properties on those child Game Objects__Animation View__.

Alternatively you can select just the child Game Object you want to animate from the Hierarchy panel or the **scene view**, and maniuplate the object or change its properties in the **inspector**, while in animation recording mode.

Using Animation Events

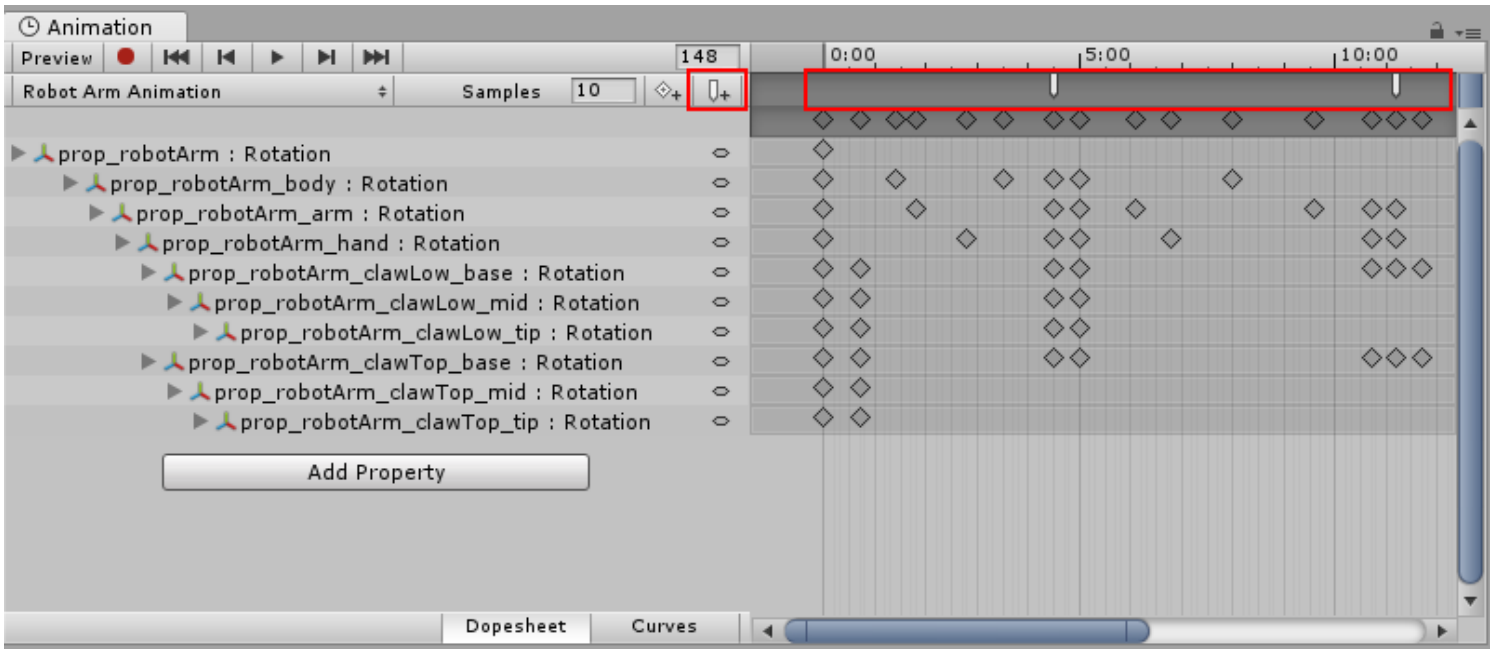
Leave feedback

You can increase the usefulness of **Animation clips** by using Animation Events, which allow you to call functions in the object's script at specified points in the **timeline**.

The function called by an Animation Event also has the option to take one parameter. The parameter can be a `float`, `string`, `int`, or `object` reference, or an AnimationEvent object. The AnimationEvent object has member variables that allow a float, string, integer and object reference to be passed into the function all at once, along with other information about the Event that triggered the function call.

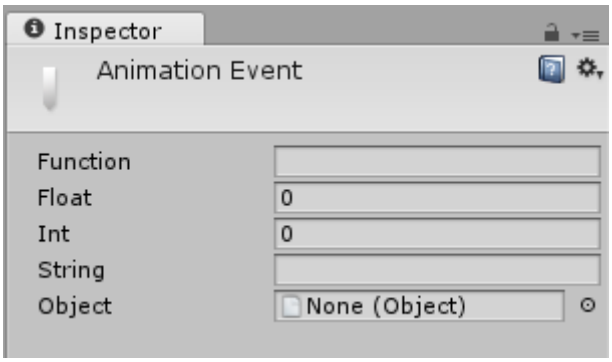
```
// This C# function can be called by an Animation Event
public void PrintFloat (float theValue) {
    Debug.Log ("PrintFloat is called with a value of " + theValue);
}
```

To add an Animation Event to a clip at the current playhead position, click the **Event** button. To add an Animation event to any point in the Animation. double-click the **Event** line at the point where you want the Event to be triggered. Once added, you can drag the mouse to reposition the Event. To delete an Event, select it and press the **Delete** key, or right-click on it and select **Delete Event**.



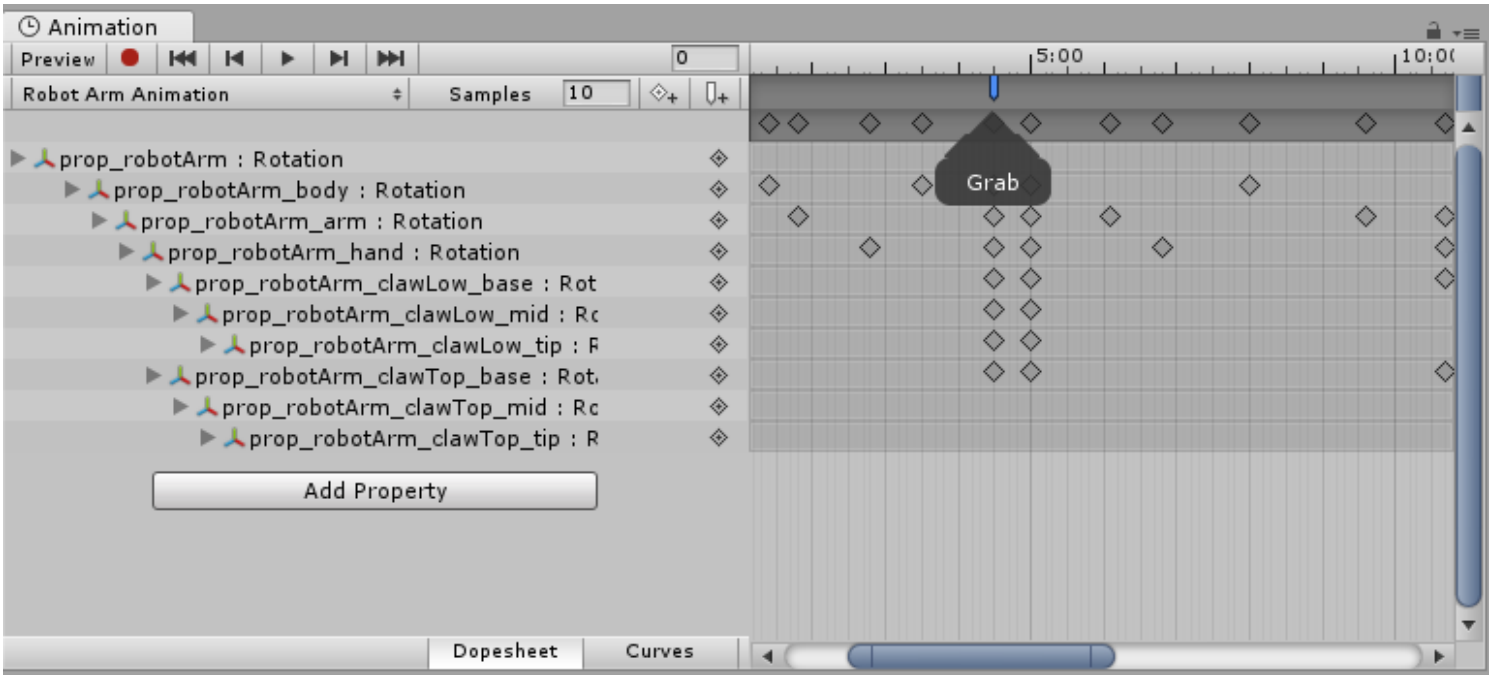
Animation Events are shown in the **Event Line**. Add a new **Animation Event** by double-clicking the **Event Line** or by using the **Event button**.

When you add an Event, the **Inspector** Window displays several fields. These fields allow you to specify the name of the function you want to call, and the value of the parameter you want to pass to it.



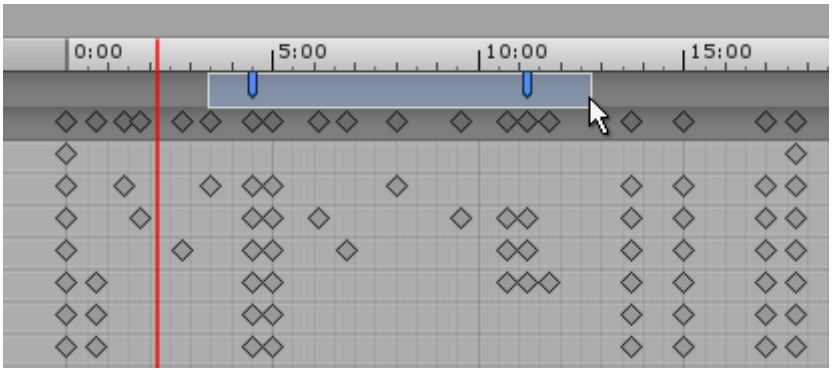
The **Animation Event** Inspector Window

The Events added to a clip are shown as markers in the Event line. Hold the mouse over a marker to show a tooltip with the function name and parameter value.



You can select and manipulate multiple Events in the timeline.

To select multiple Events in the timeline, hold the **Shift** key and select Event markers one by one to add them to your selection. You can also drag a selection box across them; click and drag within the Event marker area, like this:



Example

This example demonstrates how to add Animation Events to a simple GameObject. When all the steps are followed, the Cube animates forwards and backwards along the x-axis during Play mode, and the Event message

is displayed in the console every 1 second at the 0.8 second time.

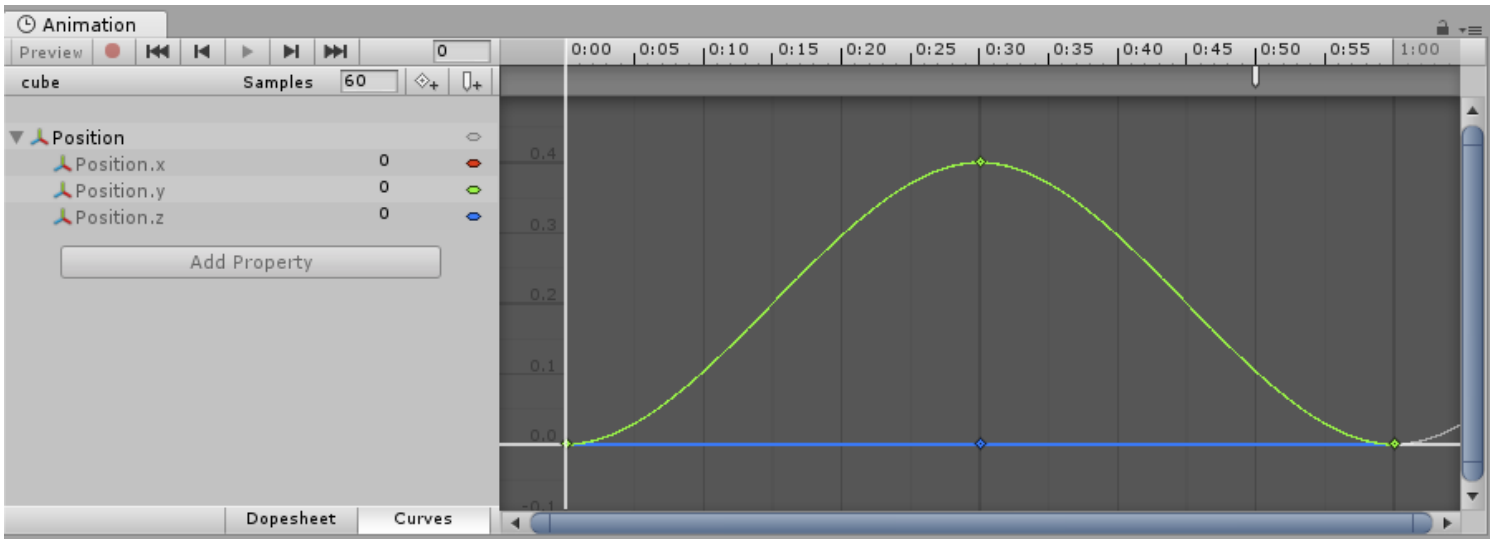
The example requires a small script with the function `PrintEvent()`. This function prints a debug message which includes a string ("called at:") and the time:

```
// This C# function can be called by an Animation Event
using UnityEngine;
using System.Collections;


public class ExampleClass : MonoBehaviour
{
    public void PrintEvent(string s)
    {
        Debug.Log("PrintEvent: " + s + " called at: " + Time.time);
    }
}
```

Create a script file with this example code and place it in your Project folder (right-click inside the Project window in Unity and select **Create** > **C# Script**, then copy and paste the above code example into the file and save it).

In Unity, create a Cube GameObject (menu: **GameObject** > **3D Object** > **Cube**). To add your new script file to it, drag and drop it from the Project window into the Inspector window.

Select the Cube and then open the Animation window (menu: **Window** > **Animation** > **Animation** or use ctrl+6). Set a **Position** curve for the x coordinate.



Animation window

Next, set the animation for the x coordinate to increase to around 0.4 and then back to zero over 1 second, then create an Animation Event at approximately 0.8 seconds. Press Play to run the animation.

2018–08–24 Page amended with limited editorial review

Animator Controllers

Leave feedback

An **Animator Controller** allows you to arrange and maintain a set of animations for a character or other animated Game Object.

The controller has references to the **animation clips** used within it, and manages the various animation states and the transitions between them using a so-called **State Machine**, which could be thought of as a kind of flow-chart, or a simple program written in a visual programming language within Unity.

The following sections cover the main features that Mecanim provides for controlling and sequencing your animations.

The Animator Controller Asset

Leave feedback

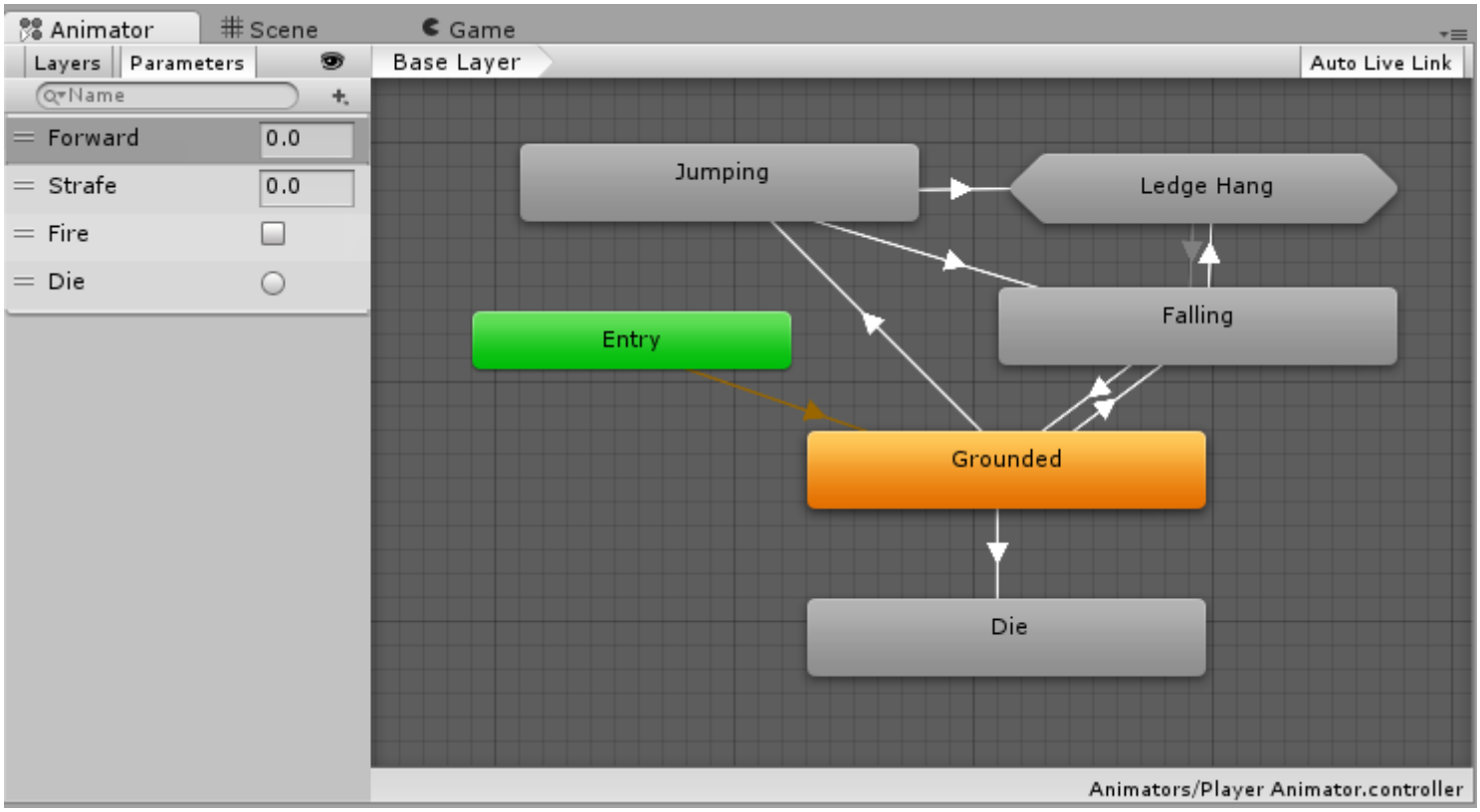
When you have an **animation clips** ready to use, you need to use an **Animator Controller** to bring them together. An Animator Controller **asset** is created within Unity and allows you to maintain a set of animations for a character or object.

An Animator Controller Asset in the Project Folder

Animator Controller assets are created from the Assets menu, or from the Create menu in the Project window.

In most situations, it is normal to have multiple animations and switch between them when certain game conditions occur. For example, you could switch from a walk animation to a jump whenever the spacebar is pressed. However even if you just have a single animation clip you still need to place it into an animator controller to use it on a Game Object.

The controller manages the various animation states and the transitions between them using a so-called **State Machine**, which could be thought of as a kind of flow-chart, or a simple program written in a visual programming language within Unity. More information about state machines can be found here. The structure of the Animator Controller can be created, viewed and modified in the Animator Window.



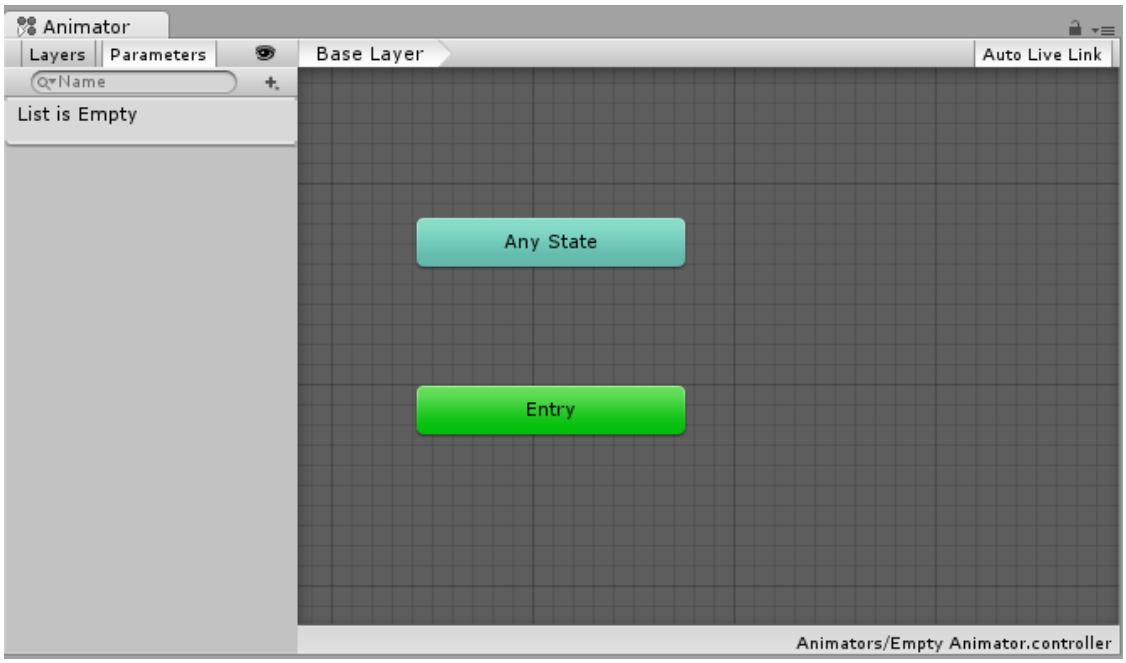
A simple Animator Controller

The animator controller is finally applied to an object by attaching an **Animator component** that references them. See the reference manual pages about the Animator component and Animator Controller for further details about their use.

The Animator Window

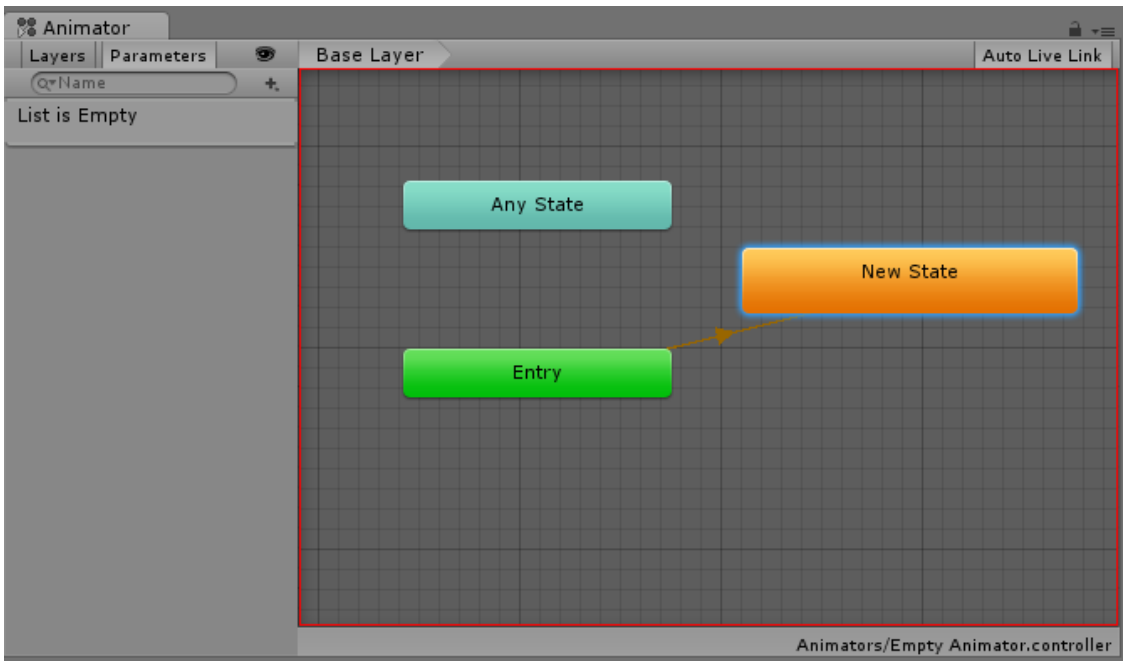
Leave feedback

The **Animator Window** allows you to create, view and modify Animator Controller **assets**.



The Animator Window showing a new empty Animator Controller asset

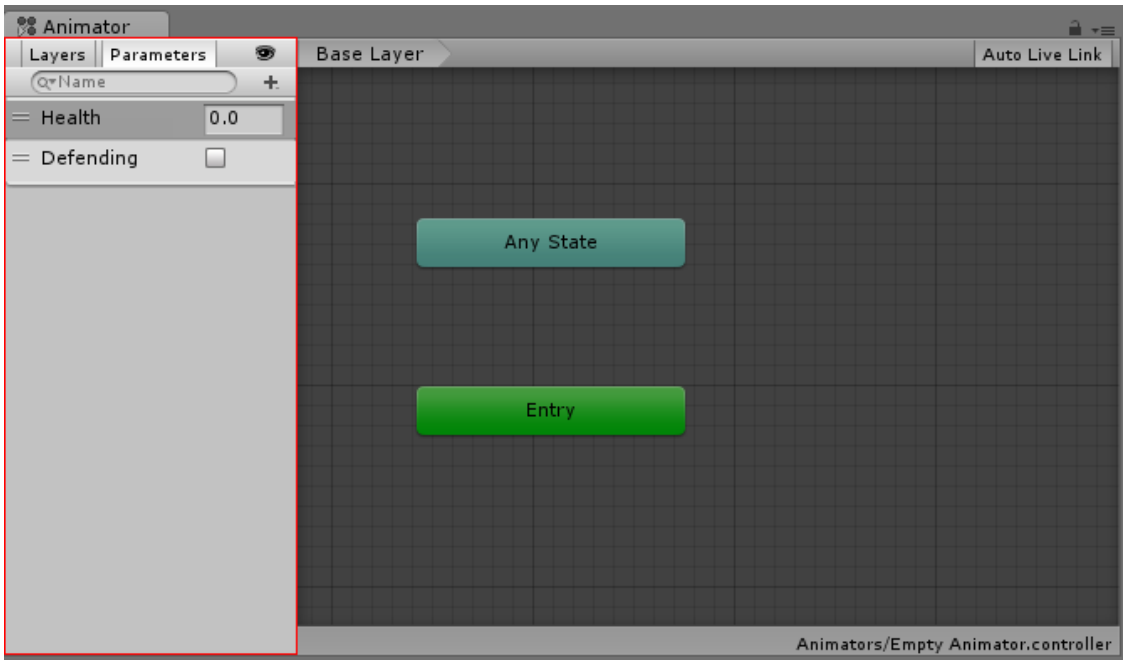
The Animator window has two main sections: the main gridded layout area, and the left-hand Layers & Parameters pane.



The layout area of the Animator window

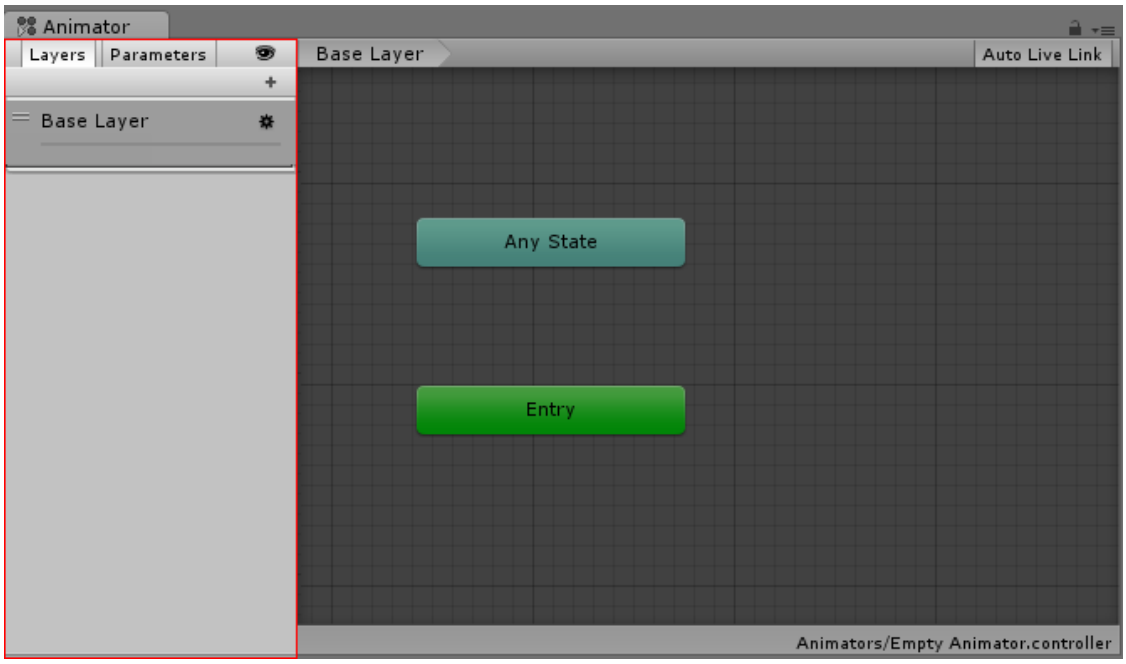
The main section with the dark grey grid is the layout area. You can use this area to create, arrange and connect states in your Animator Controller.

You can right-click on the grid to create a new state nodes. Use the middle mouse button or Alt/Option drag to pan the view around. Click to select state nodes to edit them, and click & drag state nodes to rearrange the layout of your **state machine**.



The Parameters view, with two example parameters created.

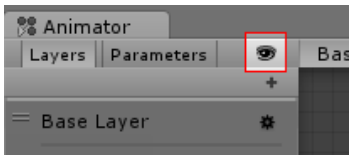
The left-hand pane can be switched betwen Parameters view and Layers view. The parameters view allows you to create, view and edit the Animator Controller Parameters. These are variables you define that act as inputs into the state machine. To add a parameter, click the Plus icon and select the parameter type from the pop up menu. To delete a parameter, select the parameter in the lists and press the delete key (on macOS use fn-Delete to delete the selected parameter).



The Layers view

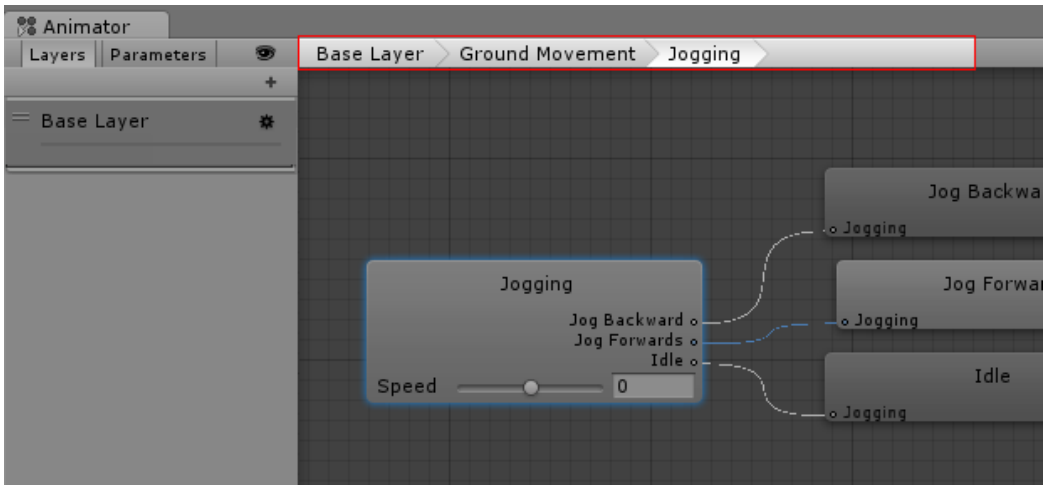
When the left-hand pane is switched to Layers view, you can create, view and edit layers within your Animator Controller. This allows you to have multiple layers of animation within a single animation controller working at the same time, each controlled by a separate state machine. A common use of this is to have a separate layer playing upper-body animations over a base layer that controls the general movement animations for a character.

To add a layer, click the plus icon. To delete a layer, select the layer and press the delete key.



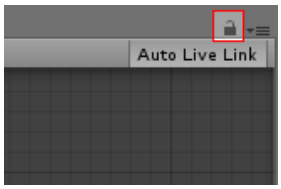
The Layers & Parameters hide icon

Clicking the “eye” icon on or off will show or hide the Parameters & Layers side-pane, allowing you more room to see and edit your state machine.



The hierarchical breadcrumb location

The “breadcrumb” hierarchical location within the current state machine. States can contain sub-states and trees and these structures can be nested repeatedly. When drilling down into sub states, the hierarchy of parent states and the current state being viewed is listed here. Clicking on the parent states allows you to jump back up to parent states or go straight back to the base layer of the state machine.



The lock icon

Enabling the lock icon will keep the Animator Window focused on the current state machine. When the lock icon is off, clicking a new animator asset or a Game Object with an animator **component** will switch the Animator Window to show that item’s state machine. Locking the window allows you to keep the Animator window showing the same state machine, regardless of which other assets or Game Objects are selected.

Animation State Machines

Leave feedback

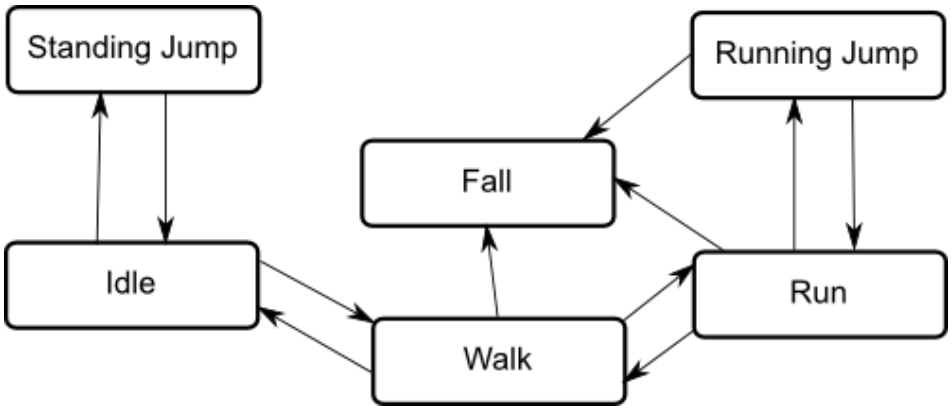
It is common for a character or other animated Game Object to have several different animations that correspond to different actions it can perform in the game. For example, a character may breathe or sway slightly while idle, walk when commanded to and raise its arms in panic as it falls from a platform. A door may have animations for opening, closing, getting jammed, and being broken open. Mecanim uses a visual layout system similar to a flow-chart, to represent a **state machine** to enable you to control and sequence the **animation clips** that you want to use on your character or object. This section gives further details about Mecanim's state machines and explains how to use them.

State Machine Basics

Leave feedback

The basic idea is that a character is engaged in some particular kind of action at any given time. The actions available will depend on the type of gameplay but typical actions include things like idling, walking, running, jumping, **etc**. These actions are referred to as **states**, in the sense that the character is in a "state" where it is walking, idling or whatever. In general, the character will have restrictions on the next state it can go to rather than being able to switch immediately from any state to any other. For example, a running jump can only be taken when the character is already running and not when it is at a standstill, so it should never switch straight from the idle state to the running jump state. The options for the next state that a character can enter from its current state are referred to as **state transitions** . Taken together, the set of states, the set of transitions and the variable to remember the current state form a **state machine**.

The states and transitions of a state machine can be represented using a graph diagram, where the nodes represent the states and the arcs (arrows between nodes) represent the transitions. You can think of the current state as being a marker or highlight that is placed on one of the nodes and can then only jump to another node along one of the arrows.

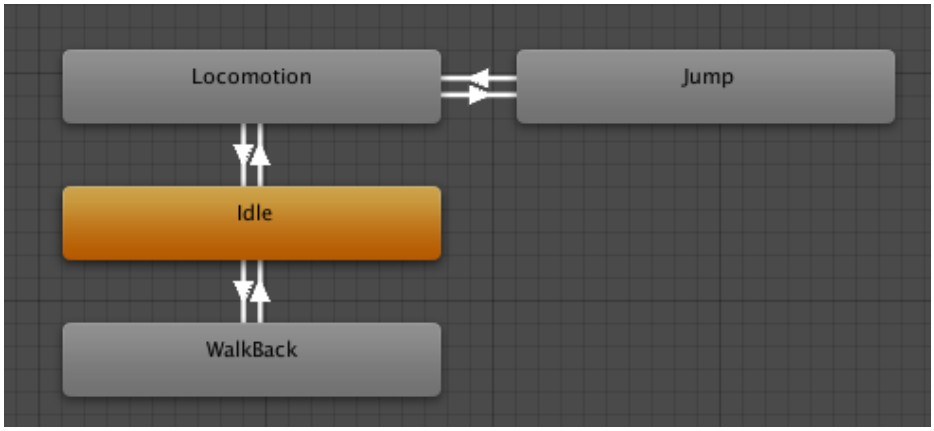


The importance of state machines for animation is that they can be designed and updated quite easily with relatively little coding. Each state has a Motion associated with it that will play whenever the machine is in that state. This enables an animator or designer to define the possible sequences of character actions and animations without being concerned about how the code will work.

State Machines

Unity's **Animation State Machines** provide a way to overview all of the **animation clips** related to a particular character and allow various events in the game (for example user input) to trigger different animations.

Animation State Machines can be set up from the Animator Controller Window, and they look something like this:



State Machines consist of **States**, **Transitions** and **Events** and smaller Sub-State Machines can be used as **components** in larger machines. See the reference pages for Animation States and Animation Transitions for further information.

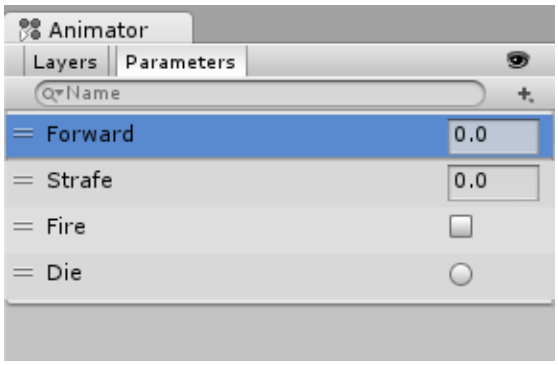
Animation Parameters

Leave feedback

Animation Parameters are variables that are defined within an **Animator Controller** that can be accessed and assigned values from **scripts**. This is how a script can control or affect the flow of the **state machine**.

For example, the value of a parameter can be updated by an animation curve and then accessed from a script so that, say, the pitch of a sound effect can be varied as if it were a piece of animation. Likewise, a script can set parameter values to be picked up by Mecanim. For example, a script can set a parameter to control a Blend Tree.

Default parameter values can be set up using the Parameters section of the **Animator window**, selectable in the top right corner of the **Animator window**. They can be of four basic types:



Int - an integer (whole number)
Float - a number with a fractional part
Bool - true or false value (represented by a checkbox)
Trigger - a boolean parameter that is reset by the controller when consumed by a transition (represented by a circle button)

Parameters can be assigned values from a script using functions in the Animator class: SetFloat, SetInt, SetBool, SetTrigger and ResetTrigger.

Here's an example of a script that modifies parameters based on user input and **collision detection**.

```
using UnityEngine;
using System.Collections;

public class SimplePlayer : MonoBehaviour {

    Animator animator;

    // Use this for initialization
    void Start () {
        animator = GetComponent<Animator>();
    }

    // Update is called once per frame
    void Update () {
        float h = Input.GetAxis("Horizontal");
        float v = Input.GetAxis("Vertical");
        bool fire = Input.GetButtonDown("Fire1");
```

```
        animator.SetFloat("Forward",v);
        animator.SetFloat("Strafe",h);
        animator.SetBool("Fire", fire);
    }

    void OnCollisionEnter(Collision col) {
        if (col.gameObject.CompareTag("Enemy"))
        {
            animator.SetTrigger("Die");
        }
    }
}
```

State Machine Transitions

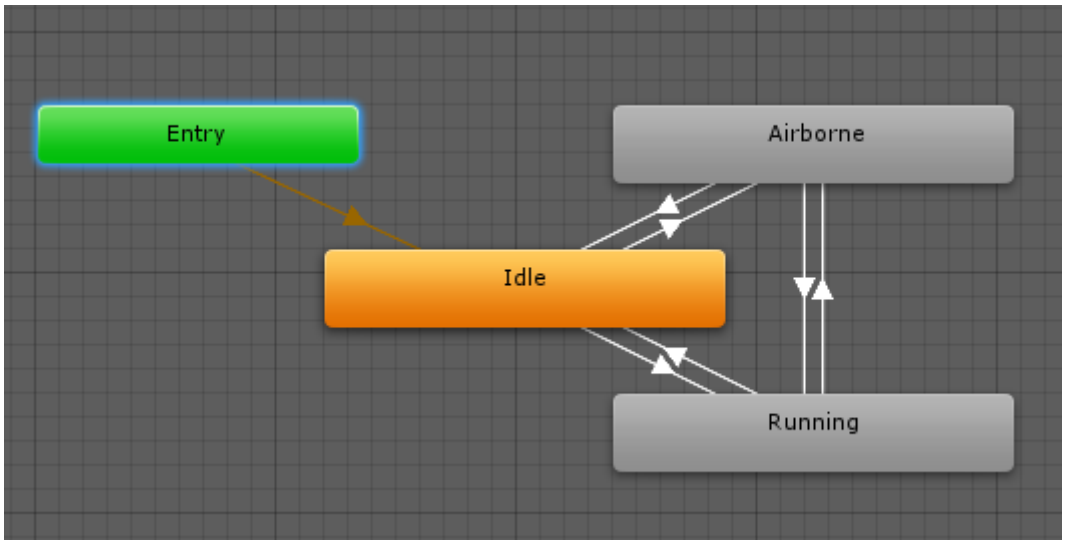
Leave feedback

State Machine Transitions exist to help you simplify large or complex State Machines. They allow you to have a higher level of abstraction over the **state machine** logic.

Each view in the **animator window** has an Entry and Exit node. These are used during State Machine Transitions.

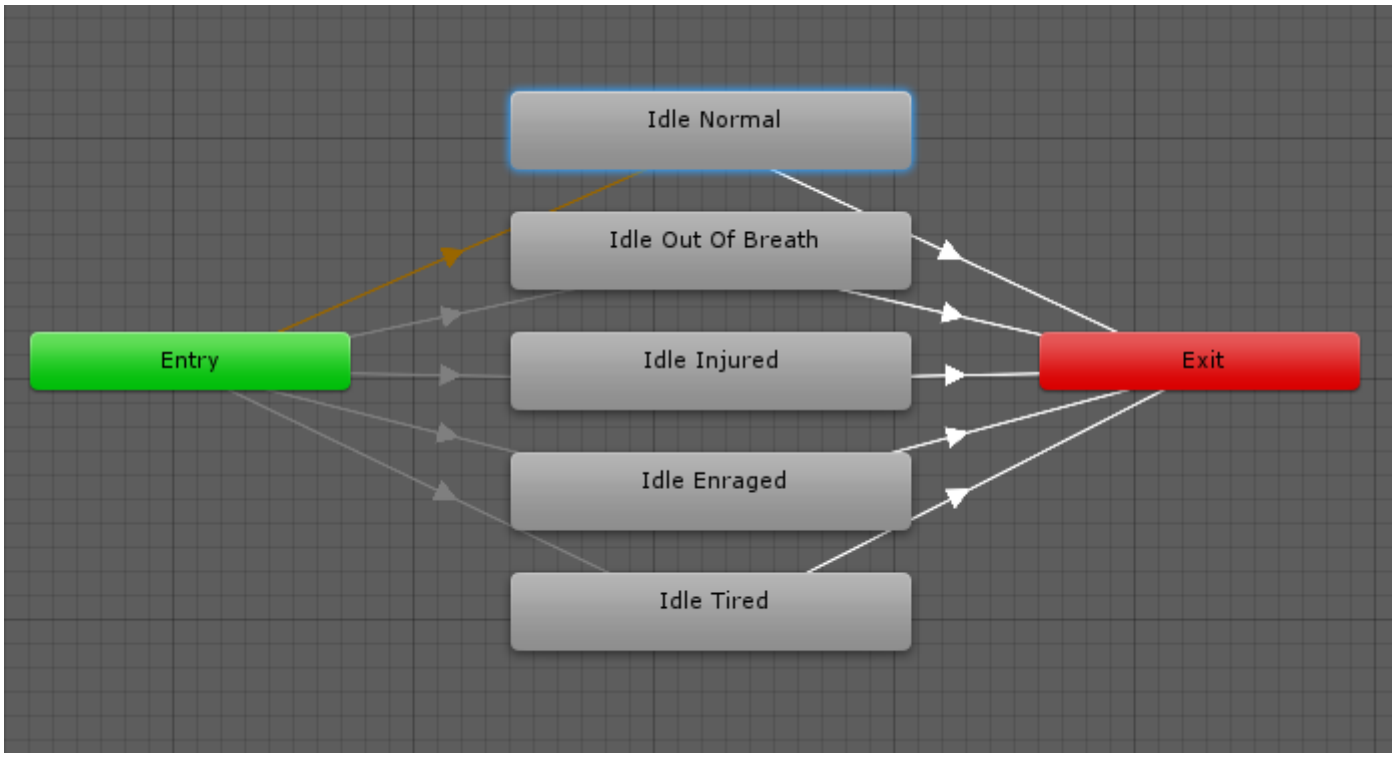
The Entry node is used when transitioning into a state machine. The entry node will be evaluated and will branch to the destination state according to the conditions set. In this way the entry node can control which state the state machine begins in, by evaluating the state of your parameters when when the state machine begins.

Because state machines always have a default state, there will always be a default transition branching from the entry node to the default state.



An entry node with a single default entry transition

You can then add additional transitions from the Entry node to other states, to control whether the state machine should begin in a different state.



An entry node with a multiple entry transitions

The Exit node is used to indicate that a state machine should exit.

Each sub-state within a state machine is considered a separate and complete state machine, so by using these entry and exit nodes, you can control the flow from a top-level state machine into its sub-state machines more elegantly.

It is possible to mix state machine transitions with regular state transtitions, so it is possible to transition from state to state, from a state to a statemachine, and from one statemachine directly to another statemachine.

State Machine Behaviours

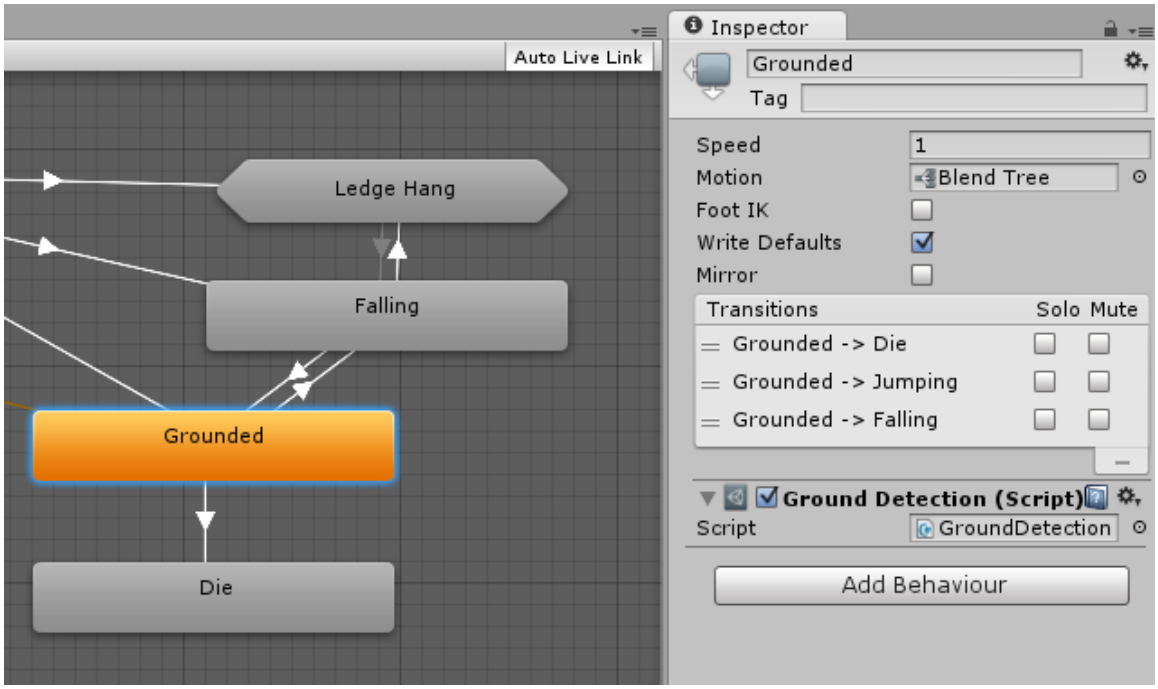
Leave feedback

A **State Machine** Behaviour is a special class of script. In a similar way to attaching regular Unity **scripts** (MonoBehaviours) to individual **GameObjects**, you can attach a StateMachineBehaviour script to an individual state within a state machine. This allows you to write code that will execute when the state machine enters, exits or remains within a particular state. This means you do not have to write your own logic to test for and detect changes in state.

A few examples for the use of this feature might be to:

Play sounds as states are entered or exited
Perform certain tests (eg, ground detection) only when in appropriate states
Activate and control special effects associated with specific states

State Machine Behaviours can be created and added to states in a very similar way to the way you would create and add scripts to GameObjects. Select a state in your state machine, and then in the **inspector** use the "Add Behaviour" button to select an existing StateMachineBehaviour or create a new one.



A state machine with a behaviour attached to the "Grounded" state

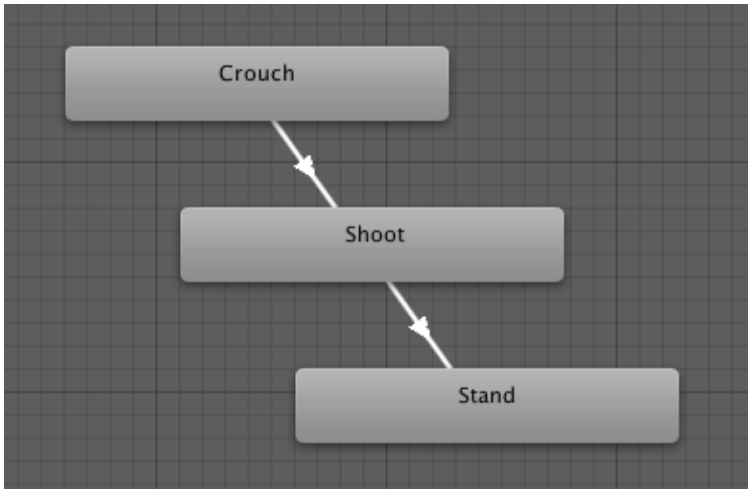
State Machine Behaviour scripts have access to a number of events that are called when the Animator enters, updates and exits different states (or sub-state machines). There are also events which allow you to handle the **Root motion** and Inverse **Kinematics** calls.

For more information see the State Machine Behaviour script reference.

Sub-State Machines

Leave feedback

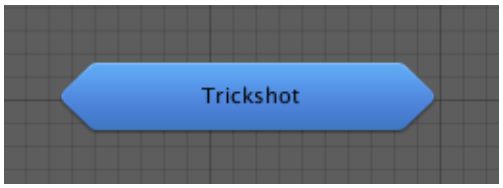
It is common for a character to have complex actions that consist of a number of stages. Rather than handle the entire action with a single state, it makes sense to identify the separate stages and use a separate state for each. For example, a character may have an action called “Trickshot” where it crouches to take a steady aim, shoots and then stands up again.



The sequence of states in a “Trickshot” action

Although this is useful for control purposes, the downside is that the **state machine** will become large and unwieldy as more of these complex actions are added. You can simplify things somewhat just by separating the groups of states visually with empty space in the editor. However, Mecanim goes a step further than this by allowing you to collapse a group of states into a single named item in the **state machine** diagram. These collapsed groups of states are called **Sub-state machines**.

You can create a sub-state machine by right clicking on an empty space within the **Animator Controller** window and selecting **Create Sub-State Machine** from the context menu. A sub-state machine is represented in the editor by an elongated hexagon to distinguish it from normal states.



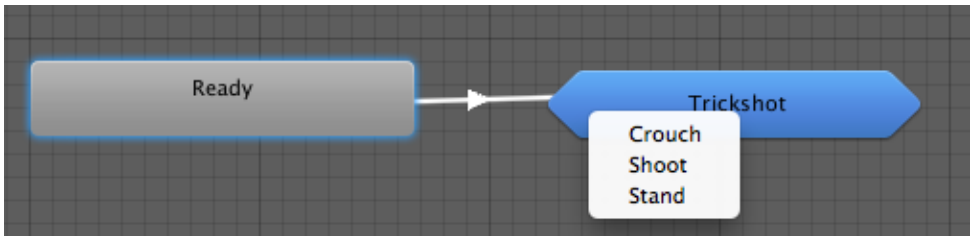
A sub-state machine

When you double-click the hexagon, the editor is cleared to let you edit the sub-state machine as though it were a completely separate state machine in its own right. The bar at the top of the window shows a “breadcrumb trail” to show which sub-state machine is currently being edited (and note that you can create sub-state machines within other sub-state machines, and so on). Clicking an item in the trail will focus the editor on that particular sub-state machine.

The “breadcrumb trail”

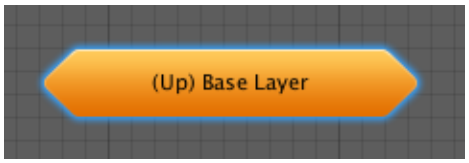
External transitions

As noted above, a sub-state machine is just a way of visually collapsing a group of states in the editor, so when you make a transition to a sub-state machine, you have to choose which of its states you want to connect to.



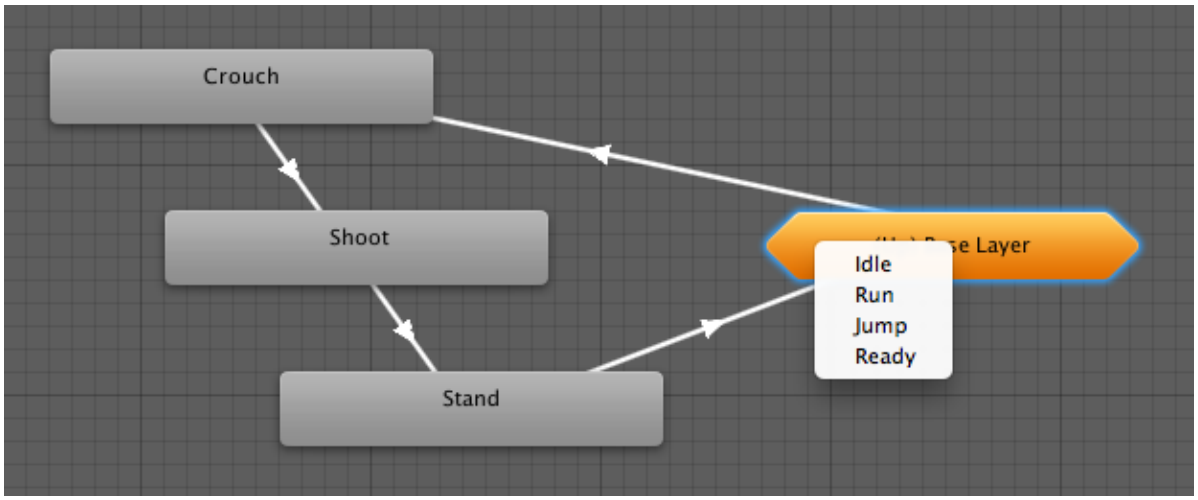
Choosing a target state within the “Trickshot” sub-state machine

You will notice an extra state in the sub-state machine whose name begins with *Up*.



The “Up” state

The *Up* state represents the “outside world”, the state machine that encloses the sub-state machine in the view. If you add a transition from a state in sub-state machine to the *Up* state, you will be prompted to choose one of the states of the enclosing machine to connect to.



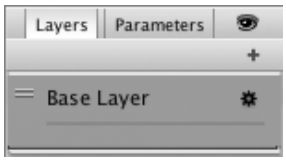
Connecting to a state in the enclosing machine

Animation Layers

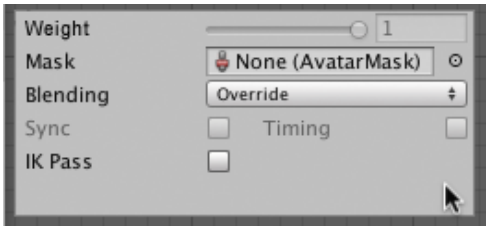
Leave feedback

Unity uses **Animation Layers** for managing complex **state machines** for different body parts. An example of this is if you have a lower-body layer for walking-jumping, and an upper-body layer for throwing objects / shooting.

You can manage animation layers from the **Layers Widget** in the top-left corner of the **Animator Controller**.



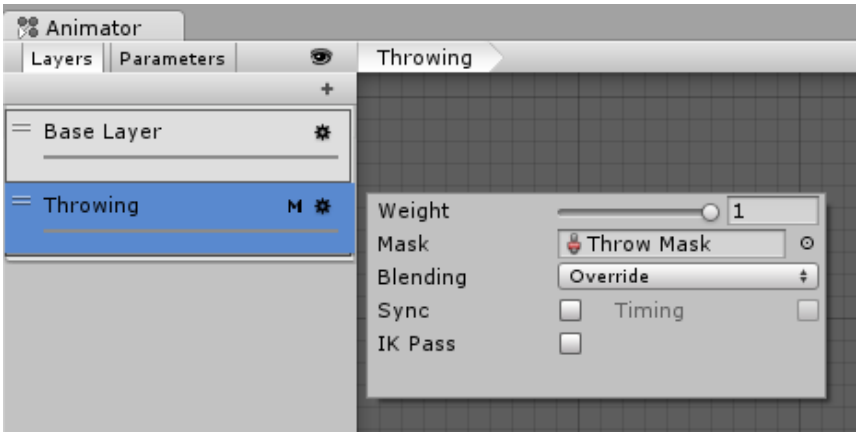
Clicking the gear wheel on the right of the window shows you the settings for this layer.



On each layer, you can specify the mask (the part of the animated model on which the animation would be applied), and the Blending type. **Override** means information from other layers will be ignored, while **Additive** means that the animation will be added on top of previous layers.

You can add a new layer by pressing the **+** above the widget.

The **Mask** property is there to specify the mask used on this layer. For example if you wanted to play a throwing animation on just the upper body of your model, while having your character also able to walk, run or stand still at the same time, you would use a mask on the layer which plays the throwing animation where the upper body sections are defined, like this:



An 'M' symbol is visible in the Layers sidebar to indicate the layer has a mask applied.

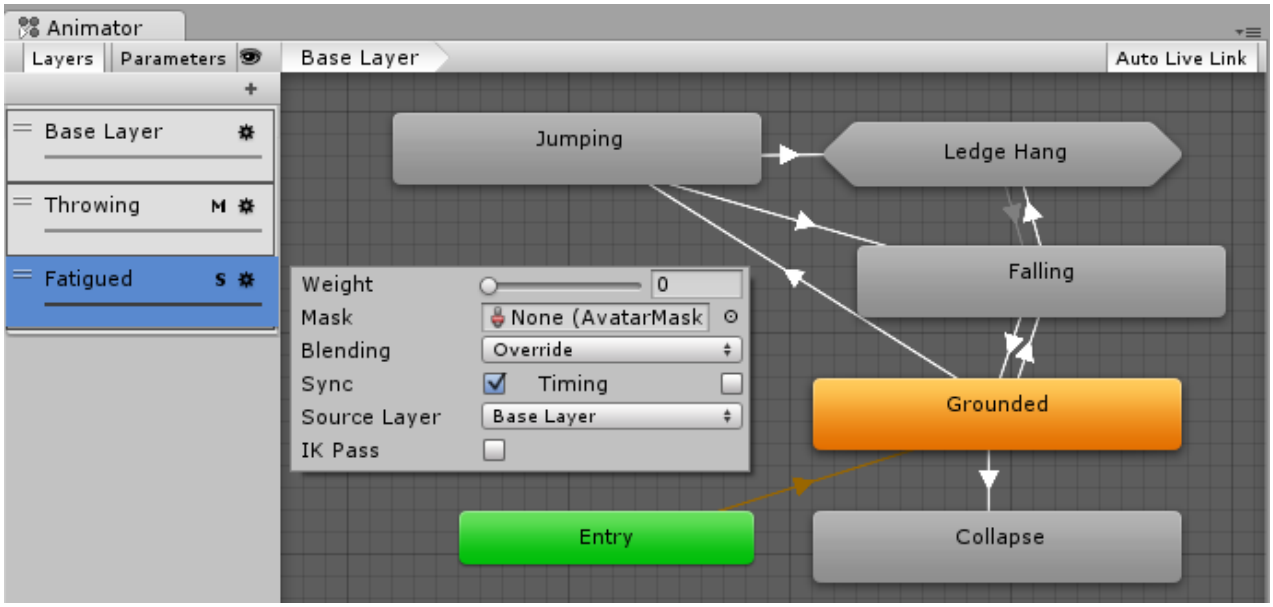
Animation Layer syncing

Sometimes it is useful to be able to re-use the same state machine in different layers. For example if you want to simulate "wounded" behavior, and have "wounded" animations for walk / run / jump instead of the "healthy" ones. You can click the **Sync** checkbox on one of your layers, and then select the layer you want to sync with. The state machine structure will then be the same, but the actual **animation clips** used by the states will be distinct.

This means the Synced layer does not have its own state machine definition at all - instead, it is an instance of the source of the synced layer. Any changes you make to the layout or structure of the state machine in the synced layer view (eg,

adding/deleting states or transitions) is done to the source of the synced layer. The only changes that are unique to the synced layer are the selected animations used within each state.

The Timing checkbox allows the animator to adjust how long each animation in synced layers takes, determined by the weight. If Timing is unselected then animations on the synced layer will be adjusted. The adjustment will be stretched to the length of the animation on the original layer. If the option is selected the animation length will be a balance for both animations, based on weight. In both cases (chosen and not chosen) the animator adjusts the length of the animations. If not chosen then the original layer is the sole master. If chosen, it is then a compromise.



In this view, the “Fatigued” layer is synced with the base layer. The state machine structure is the same as the base layer, and the individual animations used within each state are swapped for different but appropriate equivalent animations.

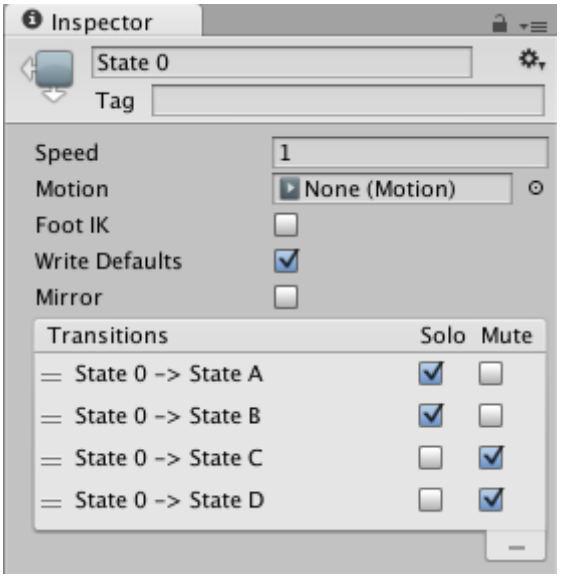
An ‘S’ is symbol is visible in the Layers sidebar to indicate the layer is a synced layer.

2018–04–25 Page amended with limited editorial review

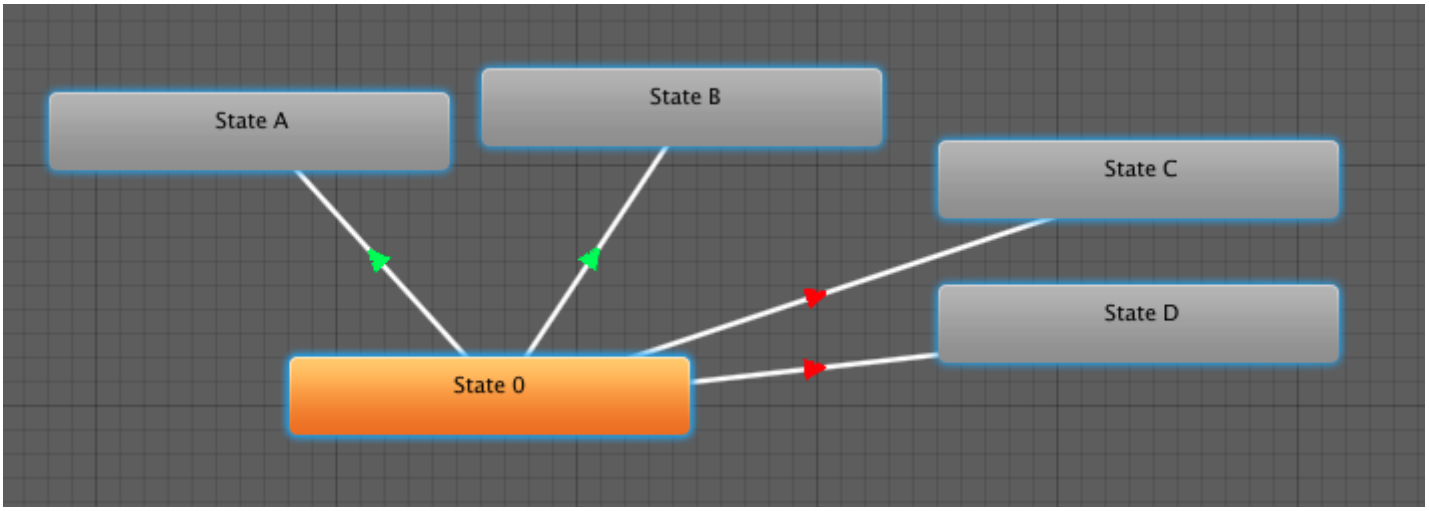
Solo and Mute functionality

Leave feedback

In complex **state machines**, it is useful to preview the operation of some parts of the machine separately. For this, you can use the Mute / Solo functionality. Muting means a transition will be disabled. Soloed transtions are enabled and with respect to other transitions originating from the same state. You can set up mute and solo states either from the **Transition Inspector**, or the **State Inspector** (recommended), where you'll have an overview of all the transitions from that state.



Soloed transitions will be shown in green, while muted transitions in red, like this:



In the example above, if you are in `State 0`, only transitions to `State A` and `State B` will be available.

The basic rule of thumb is that if one Solo is ticked, the rest of the transitions from that state will be muted.
If both Solo and Mute are ticked, then Mute takes precedence.

Known issues:

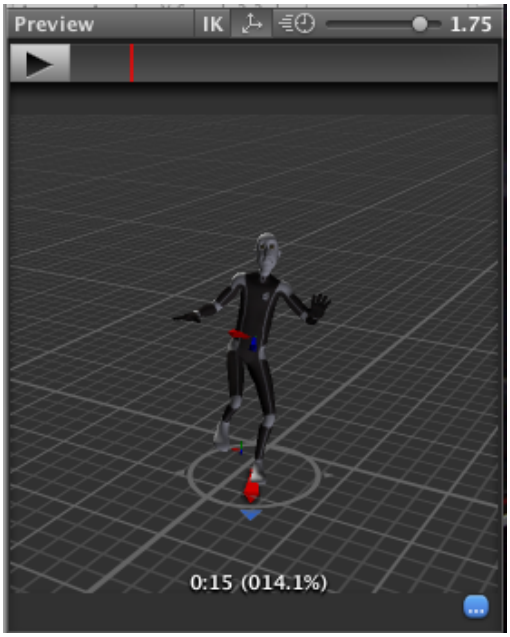
The controller graph currently doesn't always reflect the internal mute states of the engine.

Target Matching

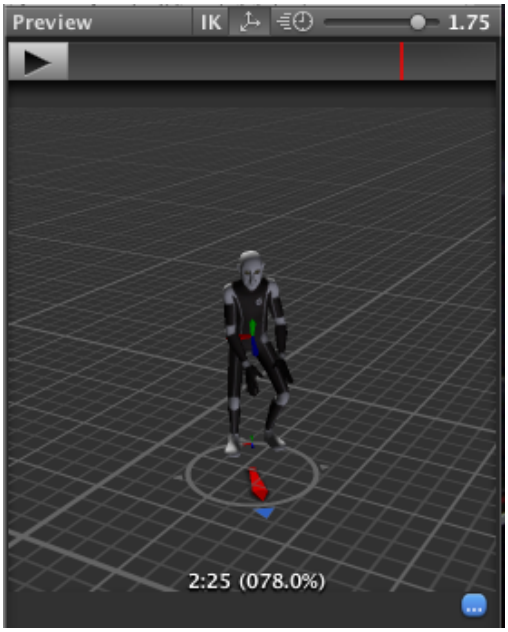
Leave feedback

Often in games, a situation arises where a character must move in such a way that a hand or foot lands at a certain place at a certain time. For example, the character may need to jump across stepping stones or jump and grab an overhead beam.

You can use the Animator.MatchTarget function to handle this kind of situation. Imagine, for example, you want to arrange an situation where the character jumps onto a platform and you already have an **animation clip** for it called *Jump Up*. Firstly, you need to find the place in the **animation clip** at which the character is beginning to get off the ground, note in this case it is 14.1% or 0.141 into the **animation clip** in normalized time:-



You also need to find the place in the animation clip where the character is about to land on its feet, which in this case is at 78.0% or 0.78.



With this information, you can create a script that calls MatchTarget which you can attach to the model:-

```
using UnityEngine;
using System;
```

```
[RequireComponent(typeof(Animator))]
public class TargetCtrl : MonoBehaviour {

    protected Animator animator;

    //the platform object in the scene
    public Transform jumpTarget = null;
    void Start () {
        animator = GetComponent<Animator>();
    }

    void Update () {
        if(animator) {
            if(Input.GetButton("Fire1"))
                animator.MatchTarget(jumpTarget.position, jumpTarget.rotation, AvatarTarg
                                                        new MatchTargetWeightMask(Vector3.
        }
    }
}
```

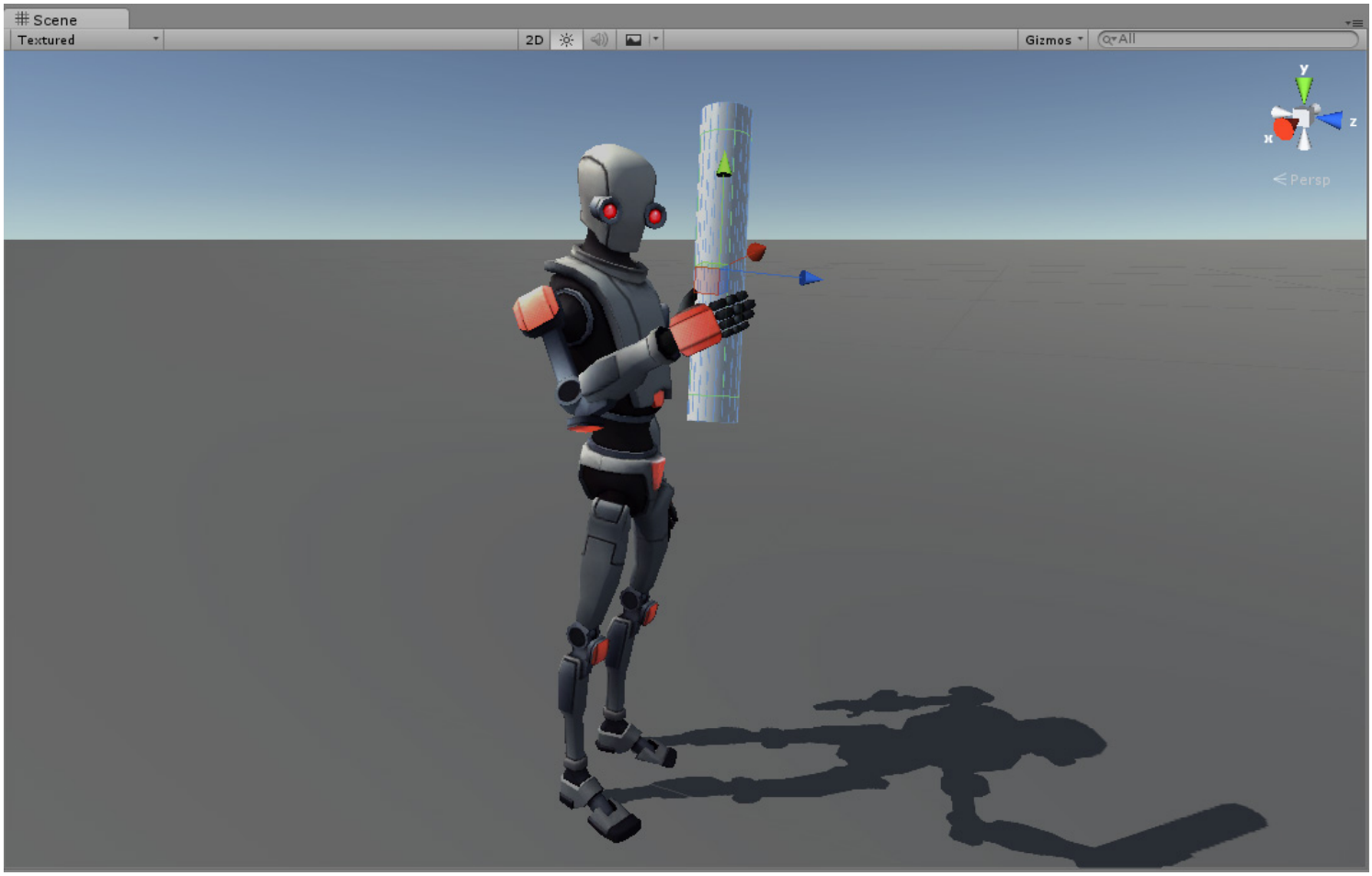
The script will move the character so that it jumps from its current position and lands with its left foot at the target. Bear in mind that the result of using MatchTarget will generally only make sense if it is called at the right point in gameplay.

Inverse Kinematics

Leave feedback

Most animation is produced by rotating the angles of **joints** in a skeleton to predetermined values. The position of a child joint changes according to the rotation of its parent and so the end point of a chain of **joints** can be determined from the angles and relative positions of the individual **joints** it contains. This method of posing a skeleton is known as **forward kinematics** .

However, it is often useful to look at the task of posing joints from the opposite point of view - given a chosen position in space, work backwards and find a valid way of orienting the joints so that the end point lands at that position. This can be useful when you want a character to touch an object at a point selected by the user or plant its feet convincingly on an uneven surface. This approach is known as **Inverse Kinematics** (IK) and is supported in Mecanim for any humanoid character *with a correctly configured Avatar*.

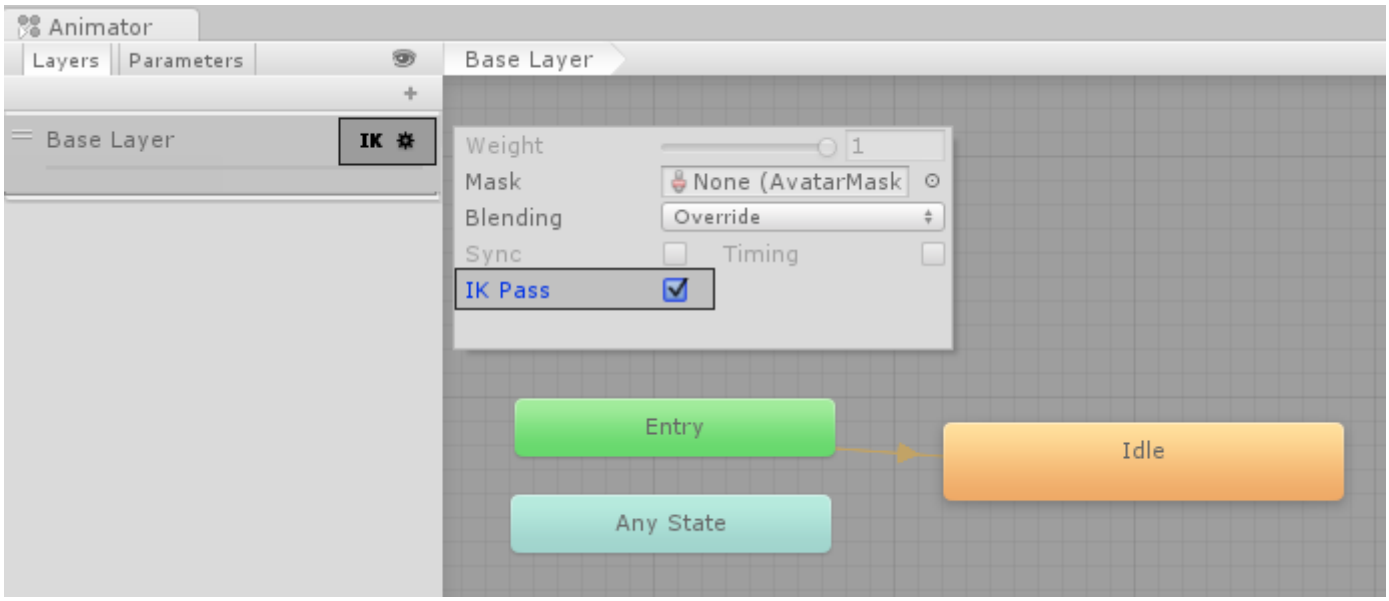


To set up IK for a character, you typically have objects around the **scene** that a character interacts with, and then set up the IK through script, in particular, Animator functions like SetIKPositionWeight, SetIKRotationWeight, SetIKPosition, SetIKRotation, SetLookAtPosition, bodyPosition, bodyRotation

In the illustration above, we show a character grabbing a cylindrical object. How do we make this happen?

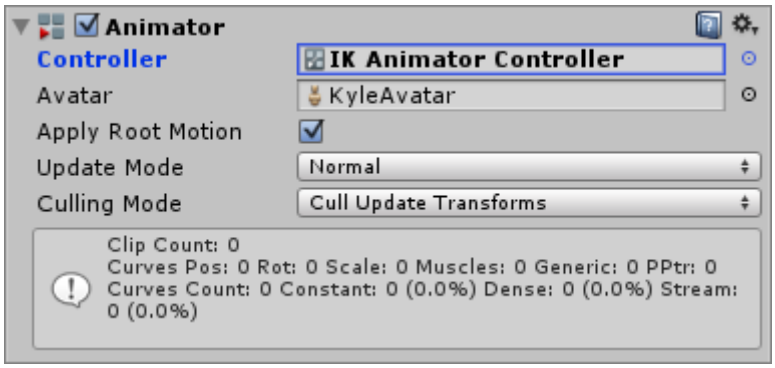
We start out with a character that has a valid **Avatar**.

Next create an **Animator Controller** with containing at least one animation for the character. Then in the Layers pane of the **Animator window**, click the cog settings icon of the Layer and and check the IK Pass checkbox in the menu which pops up.



Setting the IK Pass checkbox for the Default Layer

Make sure the Animator Controller is assigned to the character's Animator **Component**:

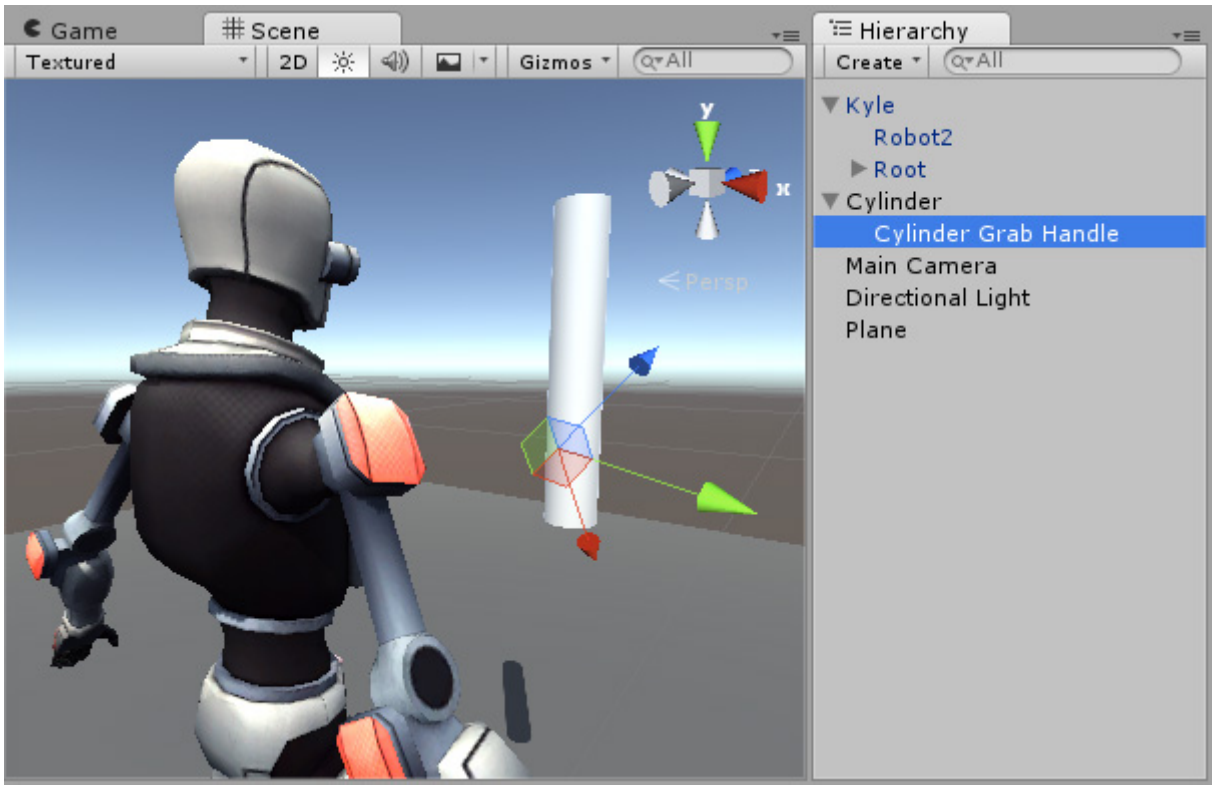


Next, attach to it a script that actually takes care of the IK, let's call it **IKControl**. This script sets the IK target for the character's right hand, and its look position to make it look at the object it is holding:

```
using UnityEngine;
using System;
using System.Collections;

[RequireComponent(typeof(Animator))]

public class IKControl : MonoBehaviour {

    protected Animator animator;

    public bool ikActive = false;
    public Transform rightHandObj = null;
    public Transform lookObj = null;

    void Start ()
    {
        animator = GetComponent<Animator>();
```

```
    }

    //a callback for calculating IK
    void OnAnimatorIK()
    {
        if(animator) {

            //if the IK is active, set the position and rotation directly to the
            if(ikActive) {

                // Set the look target position, if one has been assigned
                if(lookObj != null) {
                    animator.SetLookAtWeight(1);
                    animator.SetLookAtPosition(lookObj.position);
                }

                // Set the right hand target position and rotation, if one has b
                if(rightHandObj != null) {
                    animator.SetIKPositionWeight(AvatarIKGoal.RightHand,1);
                    animator.SetIKRotationWeight(AvatarIKGoal.RightHand,1);
                    animator.SetIKPosition(AvatarIKGoal.RightHand,rightHandObj.p
                    animator.SetIKRotation(AvatarIKGoal.RightHand,rightHandObj.r
                }

            }

            //if the IK is not active, set the position and rotation of the hand
            else {
                animator.SetIKPositionWeight(AvatarIKGoal.RightHand,0);
                animator.SetIKRotationWeight(AvatarIKGoal.RightHand,0);
                animator.SetLookAtWeight(0);
            }
        }
    }
}
```

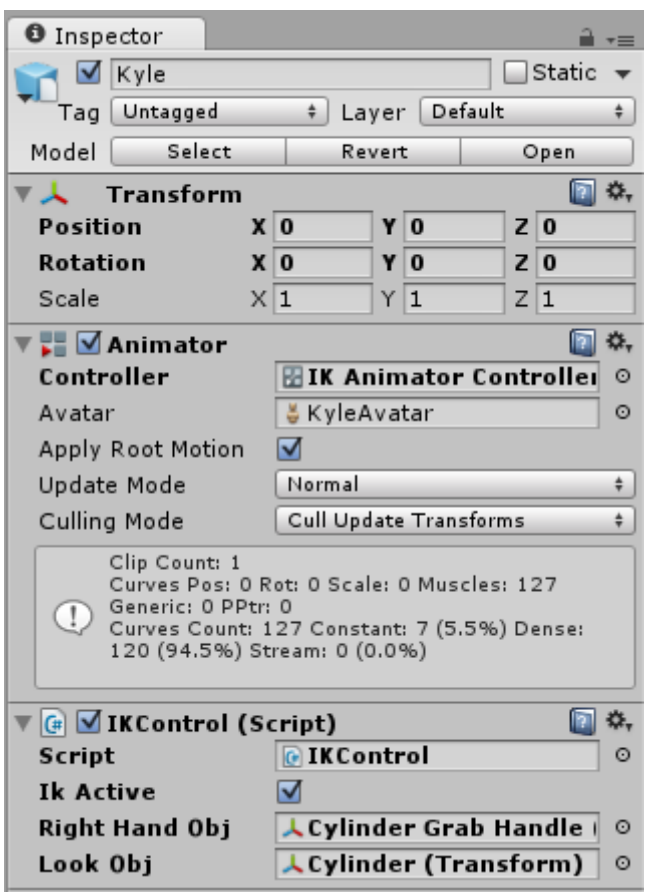
As we do not intend for the character's hand to reach inside the object to its centre (the cylinder's pivot point), we position an empty child object (in this case, named "Cylinder Grab Handle") where the hand should be on the cylinder, and rotate it accordingly. The hand then targets this child object.



An empty Game Object acts as the IK target, so the hand will sit correctly on the visible Cylinder object

This “grab handle” Game Object should then be assigned as the “Right Hand Obj” property of the IKControl script

In this example, we have the look target set to the cylinder itself, so the character looks directly towards the centre of the object even though the handle is near the bottom.



Enter play mode, and you should see the IK come to life. Observe the character grabbing and ungrabbing the object as you click the **IKActive** checkbox, and try moving the cylinder around in playmode to see the arm and hand follow the object.

Root Motion - how it works

Leave feedback

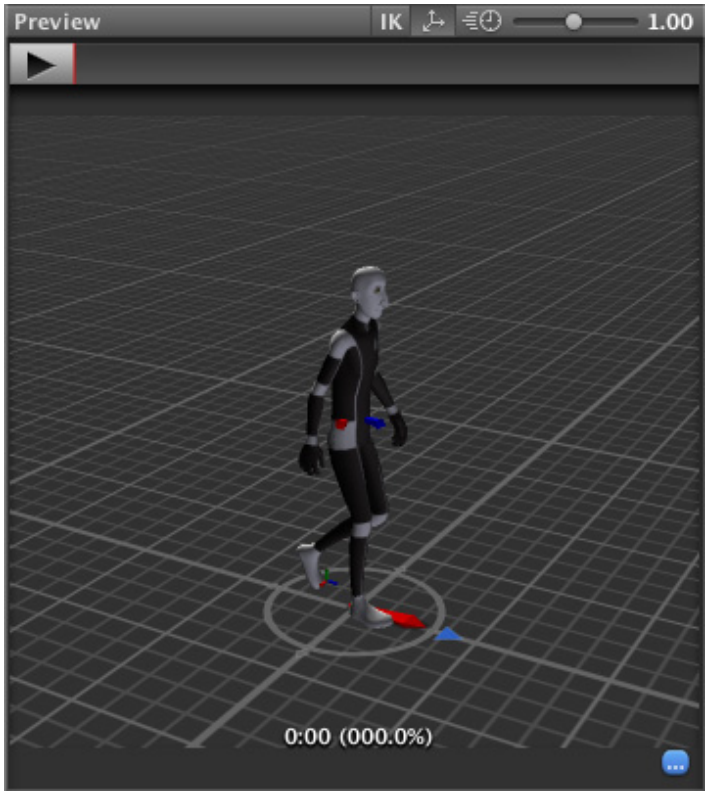
Body Transform

The Body Transform is the mass center of the character. It is used in Mecanim's **retargeting** engine and provides the most stable displacement model. The Body Orientation is an average of the lower and upper body orientation relative to the **Avatar** **T-Pose**.

The Body Transform and Orientation are stored in the **Animation Clip** (using the Muscle definitions set up in the Avatar). They are the only world-space curves stored in the **Animation Clip**. Everything else: muscle curves and IK goals (Hands and Feet) are stored relative to the body transform.

Root Transform

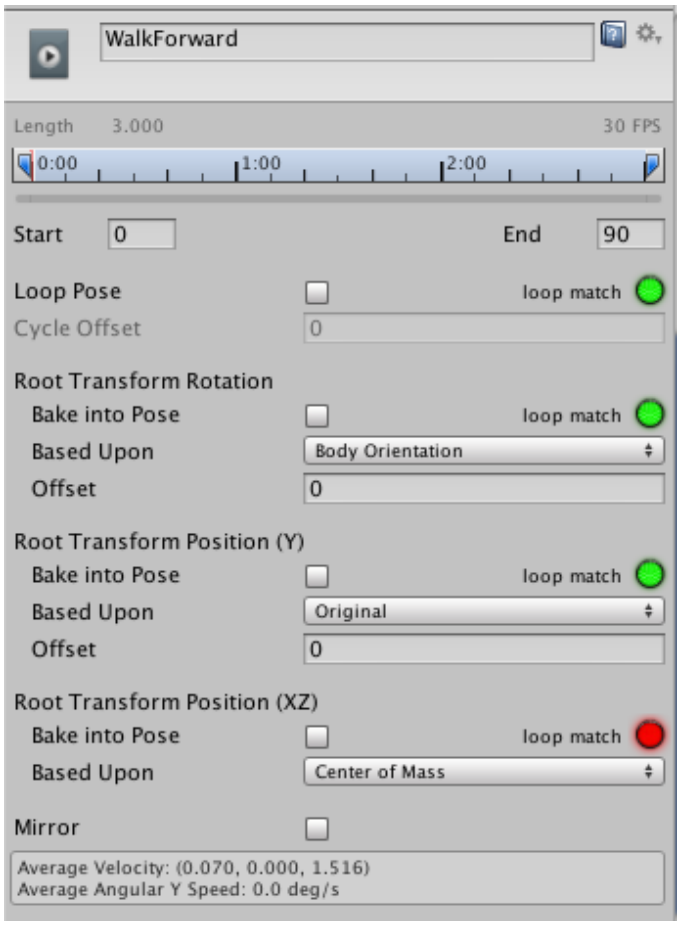
The **Root Transform** is a projection on the Y plane of the Body Transform and is computed at runtime. At every frame, a change in the **Root Transform** is computed. This change in transform is then applied to the Game Object to make it move.



The circle below the character represents the root transform

Animation Clip Inspector

The Animation Clip Editor settings - **Root Transform Rotation**, **Root Transform Position (Y)** and **Root Transform Position (XZ)** - let you control the Root Transform projection from the Body Transform. Depending on these settings some parts of the Body Transform may be transferred to Root Transform. For example you can decide if you want the motion Y position to be part of the **Root Motion** (trajectory) or part of the pose (body transform), which is known as **Baked into Pose**.



Root Transform Rotation

Bake into Pose: The orientation will stay on the body transform (or Pose). The Root Orientation will be constant and delta Orientation will be identity. This means that the Game Object will not be rotated at all by that **AnimationClip**.

Only AnimationClips that have similar start and stop Root Orientation should use this option. You will have a Green Light in the **UI** telling you that an **AnimationClip** is a good candidate. A suitable candidate would be a straight walk or a run.

Based Upon: This lets you set the orientation of the clip. Using **Body Orientation**, the clip will be oriented to follow the forward vector of body. This default setting works well for most Motion Capture (Mocap) data like walks, runs, and jumps, but it will fail with motion like strafing where the motion is perpendicular to the body's forward vector. In those cases you can manually adjust the orientation using the **Offset** setting. Finally you have **Original** that will automatically add the authored offset found in the imported clip. It is usually used with Keyframed data to respect orientation that was set by the artist.

Offset: used to enter the offset when that option is chosen for **Based Upon**.

Root Transform Position (Y)

This uses the same concepts described in Root Transform Rotation.

Bake Into Pose: The Y **component** of the motion will stay on the Body Transform (Pose). The Y component of the Root Transform will be constant and Delta Root Position Y will be 0. This means that this clip won't change the Game Object Height. Again you have a Green Light telling you that a clip is a good candidate for baking Y motion into pose.

Most of the **AnimationClips** will enable this setting. Only clips that will change the **GameObject** height should have this turned off, like jump up or down.

Note: the `Animator.gravityWeight` is driven by Bake Into Pose position Y. When enabled, `gravityWeight = 1`, when `disabled = 0`. **gravityWeight** is blended for clips when transitioning between states.

Based Upon: In a similar way to **Root Transform Rotation** you can choose from **Original** or **Mass Center (Body)**. There is also a **Feet** option that is very convenient for **AnimationClips** that change height (**Bake Into Pose** disabled). When using **Feet** the Root Transform Position Y will match the lowest foot Y for all frames. Thus the blending point always remains around the feet which prevents floating problem when blending or transitioning.

Offset: In a similar way to **Root Transform Rotation**, you can manually adjust the **AnimationClip** height using the **Offset** setting.

Root Transform Position (XZ)

Again, this uses same concepts described in **Root Transform Rotation** and **Root Motion Position (Y).**

Bake Into Pose will usually be used for "Idles" where you want to force the delta Position (XZ) to be 0. It will stop the accumulation of small deltas drifting after many evaluations. It can also be used for a Keyframed clip with **Based Upon Original** to force an authored position that was set by the artist.

Loop Pose

Loop Pose (like Pose Blending in Blend Trees or Transitions) happens in the referential of Root Transform. Once the Root Transform is computed, the Pose becomes relative to it. The relative Pose difference between Start and Stop frame is computed and distributed over the range of the clip from 0–100%.

Generic Root Motion and Loop Pose

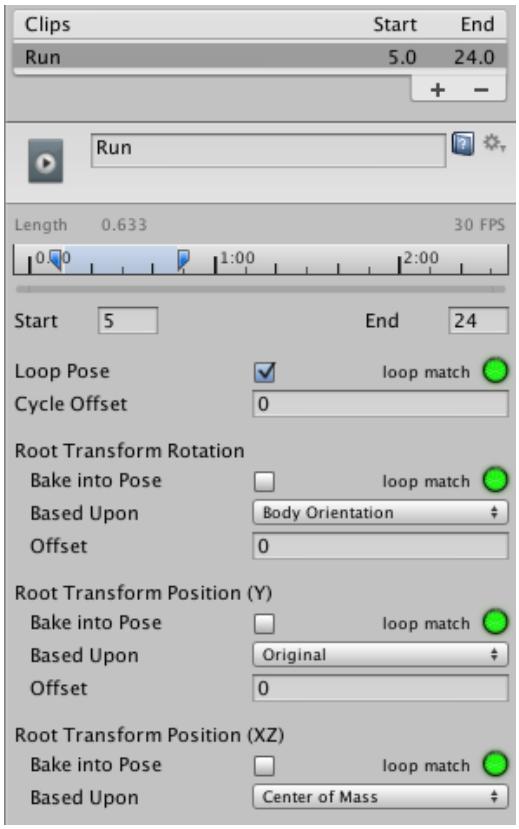
This works in essentially the same as Humanoid Root Motion, but instead of using the Body Transform to compute/project a Root Transform, the transform set in **Root Node** is used. The Pose (all the bones which transform below the Root Motion bone) is made relative to the Root Transform.

Tutorial: Scripting Root Motion for “in-place” humanoid animations

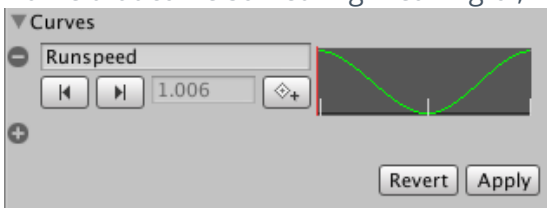
Leave feedback

Sometimes your animation comes as “in-place”, which means if you put it in a **scene**, it will not move the character that it’s on. In other words, the animation does not contain “**root motion**”. For this, we can modify **root motion** from script. To put everything together follow the steps below (note there are many variations of achieving the same result, this is just one recipe).

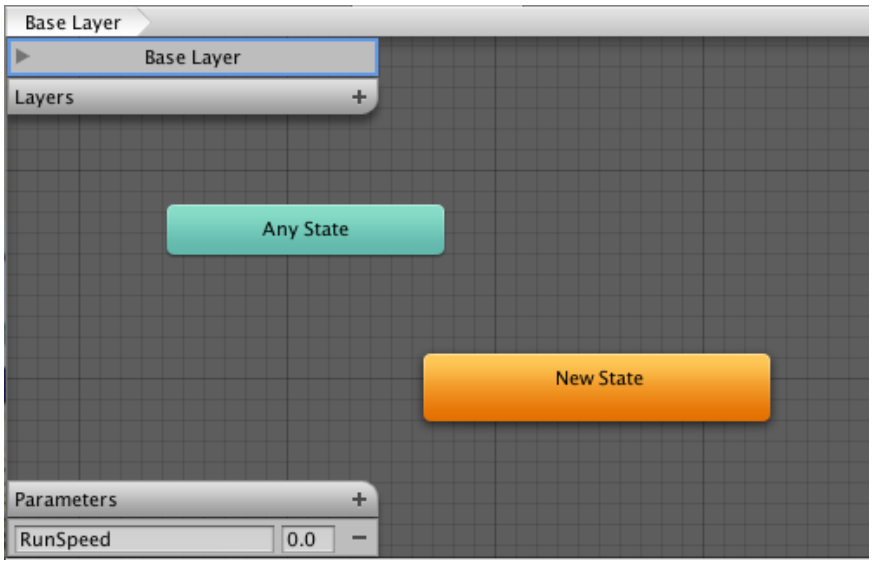
Open the **inspector** for the FBX file that contains the in-place animation, and go to the **Animation** tab
Make sure the **Muscle Definition** is set to the **Avatar** you intend to control (let’s say this avatar is called *Dude*, and it has already been added to the **Hierarchy View**).
Select the **animation clip** from the available clips
Make sure **Loop Pose** is properly aligned (the light next to it is green), and that the checkbox for **Loop Pose** is clicked



Preview the animation in the animation viewer to make sure the beginning and the end of the animation align smoothly, and that the character is moving “in-place”
On the animation clip create a curve that will control the speed of the character (you can add a curve from the **Animation Import inspector Curves-> +**)
Name that curve something meaningful, like “Runspeed”



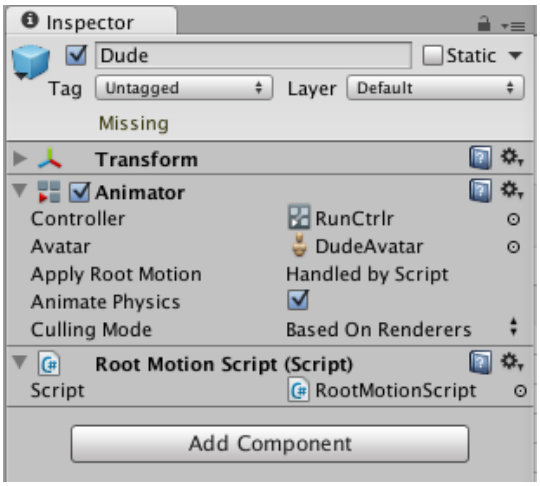
Create a new **Animator Controller**, (let’s call it **RootMotionController**)
Drop the desired animation clip into it, this should create a state with the name of the animation (say **Run**)
Add a parameter to the Controller with the same name as the curve (in this case, “Runspeed”)



Select the character **Dude** in the **Hierarchy**, whose inspector should already have an **Animator component**.
Drag **RootMotionController** onto the **Controller** property of the Animator
If you press play now, you should see the "Dude" running in place
Finally, to control the motion, we will need to create a script (RootMotionScript.cs), that implements the OnAnimatorMove callback:-

```
using UnityEngine;
using System.Collections;

[RequireComponent(typeof(Animator))]

public class RootMotionScript : MonoBehaviour {

    void OnAnimatorMove()
    {
            Animator animator = GetComponent<Animator>();

            if (animator)
            {
     Vector3 newPosition = transform.position;
               newPosition.z += animator.GetFloat("Runspeed") * Time.deltaTime;
     transform.position = newPosition;
            }
    }
}
```

You should attach RootMotionScript.cs to the "Dude" object. When you do this, the **Animator component** will detect that the script has an OnAnimatorMove function and show the **Apply Root Motion** property as *Handled by Script*


Inspector
Dude
Static
Tag
Untagged
Layer
Default
Missing
Transform
Animator
Controller
RunCtrlr
Avatar
DudeAvatar
Apply Root Motion
Handled by Script
Animate Physics
Culling Mode
Based On Renderers
Root Motion Script (Script)
Script
RootMotionScript
Add Component

Blend Trees

Leave feedback

A common task in game animation is to blend between two or more similar motions. Perhaps the best known example is the blending of walking and running animations according to the character's speed. Another example is a character leaning to the left or right as it turns during a run.

It is important to distinguish between Transitions and Blend Trees. While both are used for creating smooth animation, they are used for different kinds of situations.

Transitions are used for transitioning smoothly from one Animation State to another over a given amount of time. Transitions are specified as part of an Animation State Machine. A transition from one motion to a completely different motion is usually fine if the transition is quick.

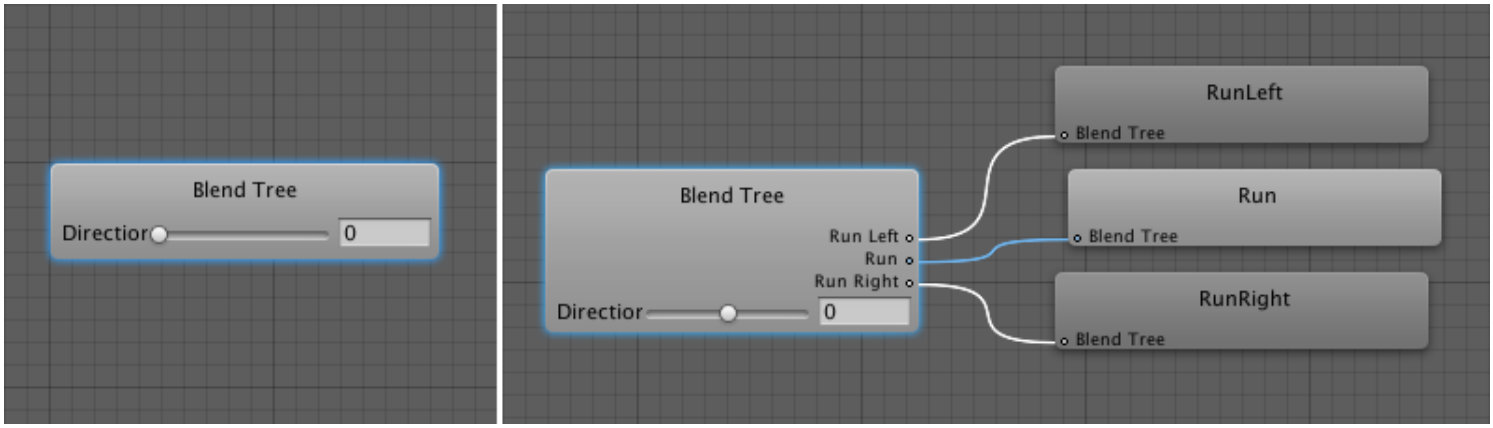
Blend Trees are used for allowing multiple animations to be blended smoothly by incorporating parts of them all to varying degrees. The amount that each of the motions contributes to the final effect is controlled using a *blending parameter*, which is just one of the numeric animation parameters associated with the **Animator Controller**. In order for the blended motion to make sense, the motions that are blended must be of similar nature and timing. Blend Trees are a special type of state in an Animation **State Machine**.

Examples of similar motions could be various walk and run animations. In order for the blend to work well, the movements in the clips must take place at the same points in normalized time. For example, walking and running animations can be aligned so that the moments of contact of foot to the floor take place at the same points in normalized time (e.g. the left foot hits at 0.0 and the right foot at 0.5). Since normalized time is used, it doesn't matter if the clips are of different length.

Using Blend Trees

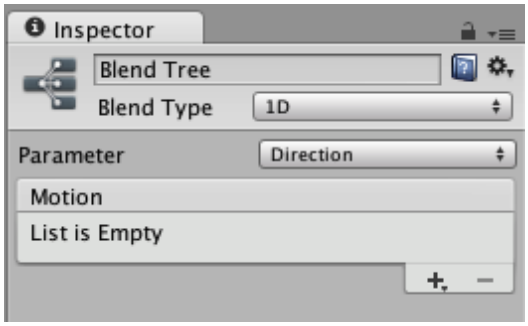
To start working with a new Blend Tree, you need to:

Right-click on empty space on the Animator Controller Window.
Select **Create State > From New Blend Tree** from the context menu that appears.
Double-click on the Blend Tree to enter the Blend Tree Graph.
The **Animator Window** now shows a graph of the entire Blend Tree while the **Inspector** shows the currently selected node and its immediate children.



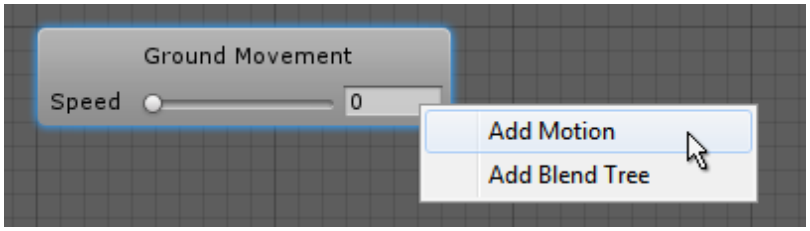
The Animator Window shows a graph of the entire Blend Tree. To the left is a Blend Tree with only the root Blend Node (no child nodes have been added yet). To the right is a Blend Tree with a root Blend Node and three Animation Clips as child nodes.

To add **animation clips** to the blend tree you can select the blend tree, then click the plus icon in the motion field in the inspector.



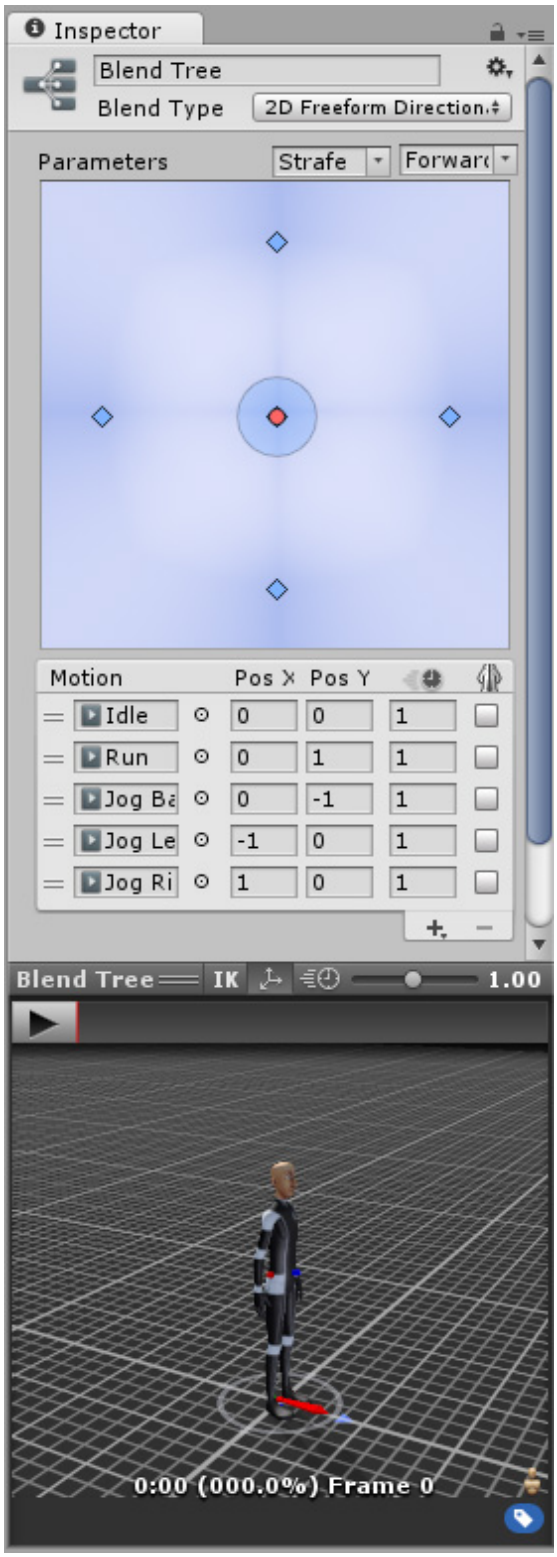
A Blend Node shown in the inspector before any motions have been added. The plus icon is used to add animation clips or child blend trees.

Alternatively, you can add animation clips or child **blend nodes** by right-clicking on the blend tree and selecting from the context menu:



The context menu when right-clicking on a blend tree node.

When the blendtree is set up with Animation clips and input parameters, the inspector window gives a graphical visualization of how the animations are combined as the parameter value changes (as you drag the slider, the arrows from the tree root change their shading to show the dominant animation clip).



A 2D Blendtree set up with five animation clips, being previewed in the inspector

You can select any of the nodes in the Blend Tree graph to inspect it in the Inspector. If the selected node is an Animation Clip the Inspector for that Animation Clip will be shown. The settings will be read-only if the animation is imported from a model. If the node is a Blend Node, the Inspector for Blend Nodes will be shown.

You can choose either 1D or 2D blending from the *Blend Type* menu; the differences between the two types are described on their own pages in this section.

Blend Trees and Root Motion

The blending between animations is handled using linear **interpolation** (ie, the amount of each animation is an average of the separate animations weighted by the blending parameter). However, you should note that root motion is *not* interpolated in the same way. See the page about root motion for further details about how this might affect your characters.

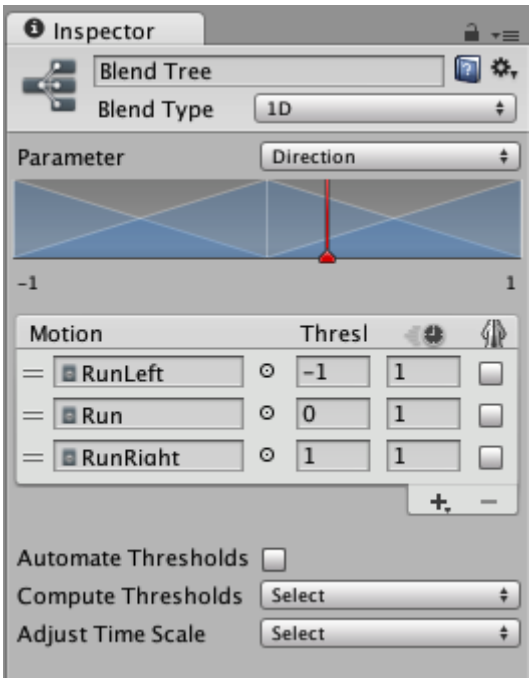
1D Blending

Leave feedback

The first option in the **Inspector** of a **Blend Node** is the **Blend Type**. This drop-down is used to select one of the different blend types that can blend according to one or two parameters. 1D Blending blends the child motions according to a single parameter.

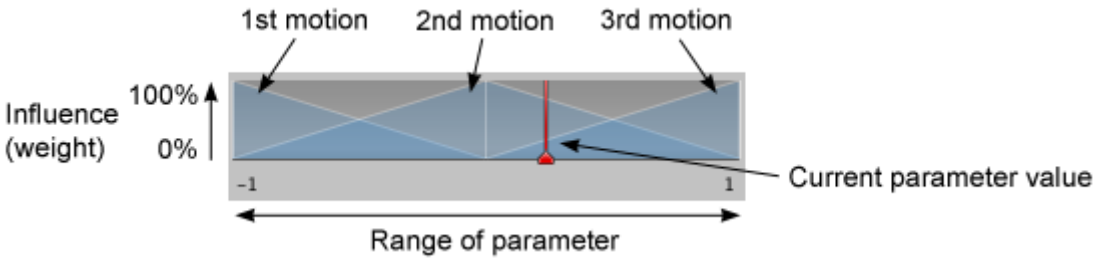
After setting the Blend Type, the first thing you need is to select the Animation Parameter that will control this Blend Tree. In this example, the parameter is *direction* which varies between –1.0 (left) and +1.0 (right), with 0.0 denoting a straight run without leaning.

Then you can add individual animations by clicking the small "+" button and selecting **Add Motion Field** from the popup menu. When you're done, it should look something like this:



A 1D Blend Tree with three Animation Clips.

The diagram at the top of the Inspector shows the influence of each of the child motions as the parameter varies between its minimum and maximum values. Each motion is shown as a little blue pyramid (the first and last are only shown in half), and if you click and hold down the left mouse button on one them, the corresponding motion is highlighted in the motion list below. The peak of each pyramid defines the parameter value where the motion has full influence, meaning that its animation weight is 1 and the other animations have a weight of 0. This is also called the **threshold** of the motion.



The diagram at the top of the Blend Tree Inspector visualizes the weights of the child motions over the range of the parameter values.

The red vertical bar indicates the value of the **Parameter**. If you press **Play** in the **Preview** at the bottom of the **Inspector** and drag the red bar in the diagram left and right, you can see how the value of the parameter is

controlling the blending of the different motions.

Parameter Range

The range of the parameter used by the Blend Tree is shown below the diagram as two numbers to the left and right. Either one of them can be changed by clicking on the number and dragging left or right with the mouse. Note that the values correspond to the threshold of the first and last motion in the motion list.

Thresholds

You can change the threshold value of a motion by clicking on its corresponding blue pyramid in the diagram and dragging it left or right. If the "Automate Thresholds" toggle is not enabled, you can also edit the threshold value of a motion in the motion list by typing in a number in the number field in the Threshold column.

Below the motion list is the checkbox *Automate Thresholds*. Enabling it will distribute the thresholds of the motions evenly across the parameter range. For example, if there are five clips and the parameter ranges from –90 to +90, the thresholds will be set to –90, –45, 0, +45 and +90 in order.

The **Compute Thresholds** drop-down will set the thresholds from data of your choice obtained from the **root motions** in the **Animation Clips**. The data that is available to choose from is speed, velocity x, y, or z, and angular speed in degrees or radians. If your **parameter** corresponds to one of these properties, you can compute the thresholds using the Compute Thresholds drop-down.

Property:	*Function:*
Speed	Sets the threshold of each motion according to its speed (the magnitude of the velocity).
Velocity X	Sets the threshold of each motion according to its velocity.x.
Velocity Y	Sets the threshold of each motion according to its velocity.y.
Velocity Z	Sets the threshold of each motion according to its velocity.z.
Angular Speed (Rad)	Sets the threshold of each motion according to its angular speed in radians per second.
Angular Speed (Deg)	Sets the threshold of each motion according to its angular speed in degrees per second.

Say, for example, you had a walk animation that covered 1.5 units per second, a jog at 2.3 units per second, and a run at 4 units per second, choosing the **Speed** option from the drop-down would set the parameter range and thresholds for the three animations based on these values. So, if you set the speed parameter to 3.0, it would blend the jog and run with a slight bias toward the jog.

2D Blending

Leave feedback

The first option in the **Inspector** of a **Blend Node** is the **Blend Type**. This drop-down is used to select one of the different blend types that can blend according to one or two parameters. The 2D blending types blends the child motions according to two parameters.

The different 2D Blend Types have different uses that they are suitable for. They differ in how the influence of each motion is calculated.

2D Simple Directional: Best used when your motions represent different directions, such as “walk forward”, “walk backward”, “walk left”, and “walk right”, or “aim up”, “aim down”, “aim left”, and “aim right”. Optionally a single motion at position (0, 0) can be included, such as “idle” or “aim straight”. In the Simple Directional type there should *not* be multiple motions in the same direction, such as “walk forward” and “run forward”.

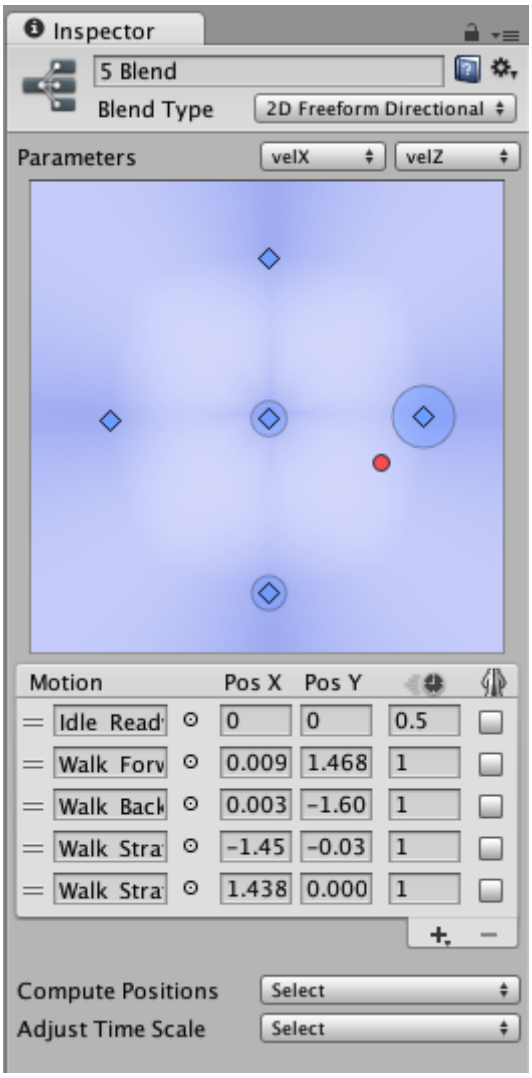
2D Freeform Directional: This blend type is also used when your motions represent different directions, however you can have multiple motions in the same direction, for example “walk forward” and “run forward”. In the Freeform Directional type the set of motions should always include a single motion at position (0, 0), such as “idle”.

2D Freeform Cartesian: Best used when your motions do not represent different directions. With Freeform Cartesian your X parameter and Y parameter can represent different concepts, such as angular speed and linear speed. An example would be motions such as “walk forward no turn”, “run forward no turn”, “walk forward turn right”, “run forward turn right” **etc**.

Direct: This type of blend tree lets user control the weight of each node directly. Useful for facial shapes or random idle blending.

After setting the Blend Type, the first thing you need is to select the two Animation Parameters that will control this Blend Tree. In this example, the parameters are *velocityX* (strafing) and *velocityZ* (forward speed).

Then you can add individual animations by clicking **+ -> Add Motion Field** to add an Animation Clip to the blend tree. When you’re done, it should look something like this:

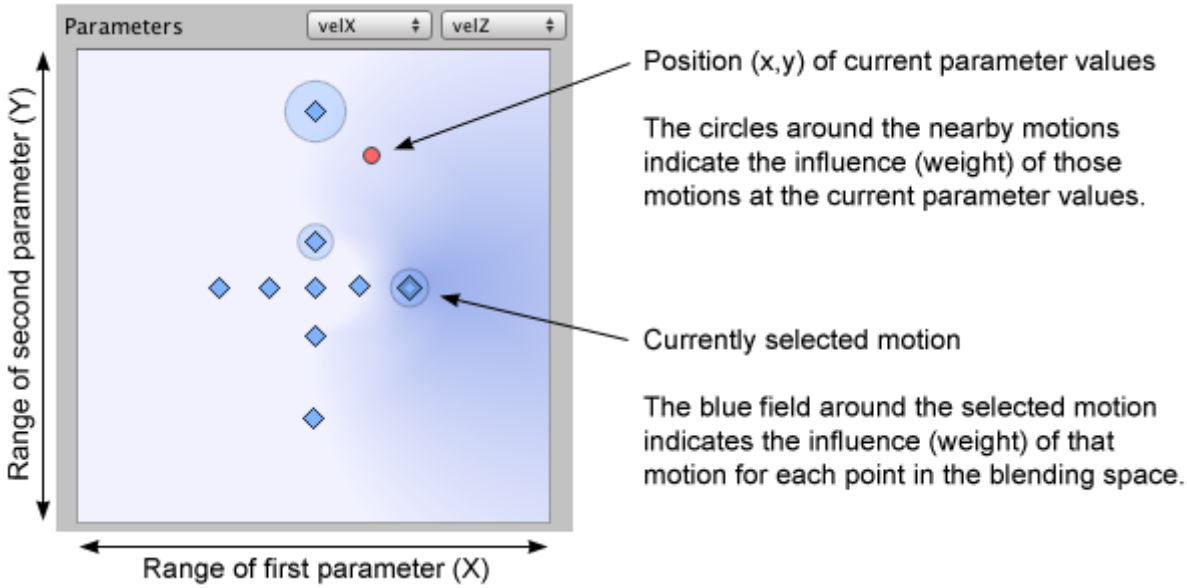


A 2D Blend Node with five Animation Clips.

The positions in 2D blending are like the thresholds in 1D blending, except that there are two values instead of one, corresponding to each of the two parameters. Their positions along the horizontal X axis correspond to the first parameter, and their positions along the vertical Y axis correspond to the second parameter. A walking forward animation might have a velocityX of 0 and a velocityZ of 1.5, so those values should be typed into the Pos X and Pos Y number fields for the motion.

The 2D Blending Diagram

The diagram at the top of the Inspector shows the positions of the child motions in the 2D blend space. The motions are shown as blue dots. Motions with no **Animation Clip** or Blend Tree assigned have no influence on the blend and are shown as gray dots. You can select a motion by clicking on its dot in the diagram. Once selected, the influence of that motion for each point in the blending space is visualized as a blue field. The field is strongest right under the position of the motion, where the motion has full influence, meaning that its animation weight is 1 and the other animations have a weight of 0. Further away the influence decreases as the influence of other motions take over.



The diagram at the top of the Blend Node Inspector visualizes the weights of the child motions over the extends of the parameter values.

The red dot indicates the values of the two **Parameters**. If you press **Play** in the **Preview** at the bottom of the **Inspector** and drag the red dot in the diagram around, you can see how the values of the parameters are controlling the blending of the different motions. In the diagram you can also see the influence of each motion represented as circles around each motion. You will see that if you move the red dot on top of one of the blue dots representing a motion, the circle for that motion gains its maximum radius and the circles for all other motions shrink down to nothing. At positions that are in between several motions, multiple of the nearby motions will have an influence on the blend. If you select one of the motions in order to see the blue influence field of that motion, you can see that as you move the red dot around, the circle size of the motion corresponds exactly with how strong the influence field is at various positions.

When no motion is selected, the diagram shows a mix of all the influence fields that is more blue where a single motion dominates and less blue where many motions contribute to the blend.

Positions

You can change the positions of a motion by clicking on its corresponding blue dot in the diagram and dragging it around. You can also edit position coordinates of a motion in the motion list by typing in numbers in the number fields in the Pos X and Pos Y columns.

The **Compute Positions** drop-down will set the positions from data of your choice obtained from the **root motions** in the Animation Clips. The data that is available to choose from is speed, velocity x, y, or z, and angular speed in degrees or radians. If one or both of your **parameters** correspond to one of these properties, you can compute the Pos X and/or Pos Y using the Compute Positions drop-down.

Property:	*Function:*
Velocity XZ	Sets the Pos X of each motion according to its velocity.x and the Pos Y according to its velocity.z.
Speed And Angular Speed	Sets the Pos X of each motion according to its angular speed (in radians per second) and the Pos Y according to its speed.

Furthermore you can mix and match by choosing **Compute Position -> X Position From** and/or **Compute Position -> Y Position From** to only auto-compute one of them at a time, leaving the other unchanged.

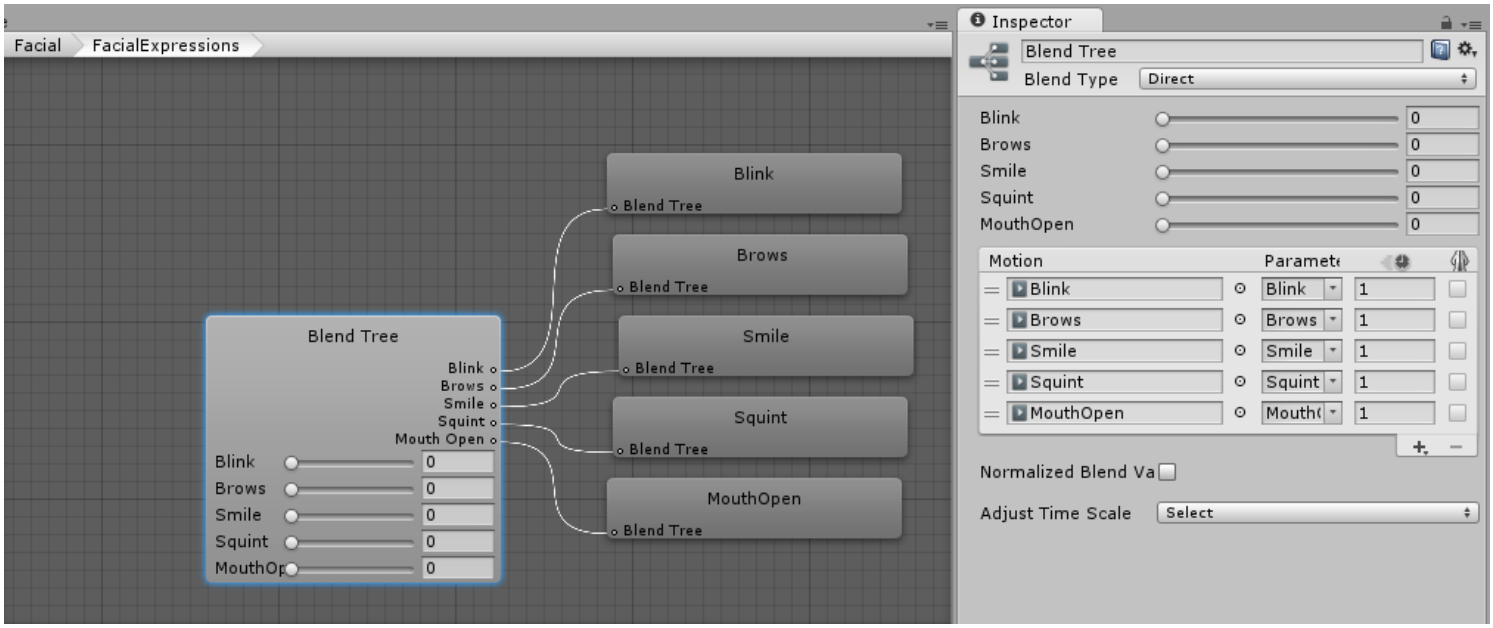
Property:	*Function:*
Speed	Sets the Pos X or Pos Y of each motion according to its speed (the magnitude of the velocity).
Velocity X	Sets the Pos X or Pos Y of each motion according to its velocity.x.
Velocity Y	Sets the Pos X or Pos Y of each motion according to its velocity.y.
Velocity Z	Sets the Pos X or Pos Y of each motion according to its velocity.z.
Angular Speed (Rad)	Sets the Pos X or Pos Y of each motion according to its angular speed in radians per second.
Angular Speed (Deg)	Sets the Pos X or Pos Y of each motion according to its angular speed in degrees per second.

Say, for example, that your parameters correspond to sideways velocity and forward velocity, and that you have an idle animation with an average velocity (0, 0, 0), a walk animation with (0, 0, 1.5), and two strafe animations with velocities of (–1.5, 0, 0) and (1.5, 0, 0) respectively. Choosing the *Velocity XZ* option from the drop-down would set the positions of the motions according to the X and Z coordinates of those velocities.

Direct Blending

Leave feedback

Using a **Direct Blend Tree** allows you to map animator parameters directly to the weight of a BlendTree child. This can be useful if you want to have exact control over the various animations that are being blended rather than blend them indirectly using one or two parameters (in the case of 1D and 2D blend trees).

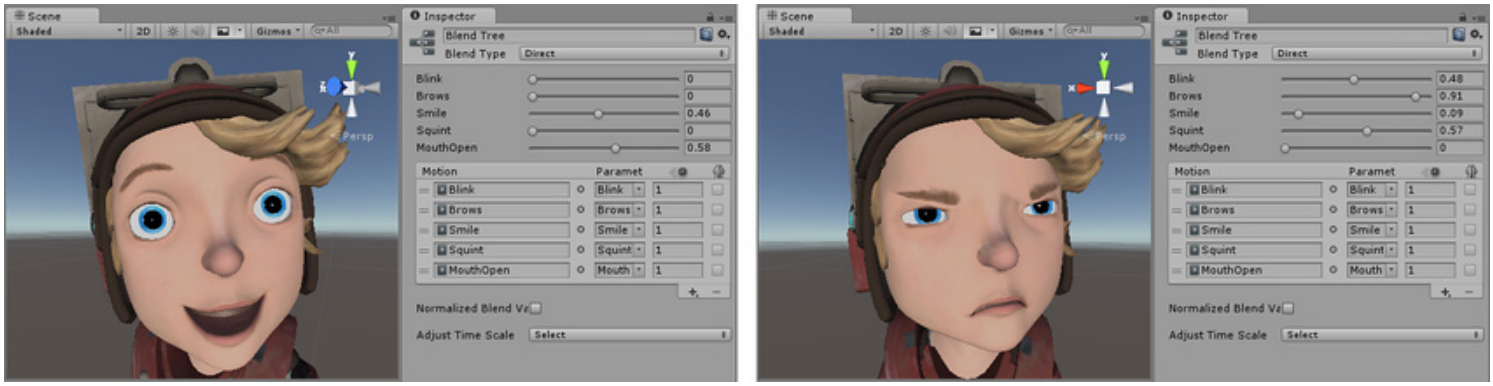


A Direct Blend Tree with five animation clips assigned.

When setting up a direct blend tree, the **inspector** allows you to add motions to the motion list. Each motion should then be assigned a corresponding parameter to directly control its blend weight in the tree. Read more about creating Animator Parameters here.

In effect, this Direct mode simply bypasses the crossfading, or the various 2D blending algorithms (Freeform Directional, Freeform Cartesian, etc) and allows you to implement whatever code you like to control the mix of blended animations.

This can be particularly useful when mixing blendshape animations for facial expressions, or when blending together additive animations.



The blend weights for each clip can be blended arbitrarily.

Additional Blend Tree Options

Leave feedback

The options below are common to both 1D and 2D blending.

Time Scale

You can alter the “natural” speed of the animation clips using the animation speed number fields (the columns with a clock icon at the top), so you could make the walk twice as fast by using a value of 2.0 as its speed. The **Adjust Time Scale > Homogeneous Speed** button rescales the speeds of the clips so that they correspond with the chosen minimum and maximum values of the parameter but keep the same *relative* speeds they initially had.

Note that the **Adjust Time Scale** drop-down is only available if all the motions are **Animation Clips** and not child Blend Trees.

Mirroring

You can mirror any **humanoid Animation** Clip in the motions list by enabling the mirror toggle at the far right. This feature enables you to use the same animation in its original form and in a mirrored version without needing twice the memory and space.

Animation Blend Shapes

Leave feedback

Preparing the Artwork

Once you have your Blend Shapes set up in Autodesk® Maya®:

Export your selection to fbx ensuring the animation box is checked and blend Shapes under deformed models is checked.

Import your FBX file into Unity (from the main Unity menu: **Assets** > **Import New Asset** and then choose your file).

Drag the Asset into the hierarchy window. If you select your object in the hierarchy and look in the **inspector**, you will see your Blend Shapes are listed under the SkinnedMeshRenderer **component**. Here you can adjust the influence of the blend shape to the default shape, 0 means the blend shape has no influence and 100 means the blend shape has full influence.

Create Animations In Unity

It is also possible to use the Animation window in Unity to create a blend animation, here are the steps:

Open the Animation window under **Window** > **Animation** > **Animation**.

On the left of the window click 'Add Curve' and add a Blend Shape which will be under Skinned **Mesh** Renderer.

From here you can manipulate the **keyframes** and Blend Weights to create the required animation.

Once you are finished editing your animation you can click play in the editor window or the animation window to preview your animation.

Scripting Access

It's also possible to set the blend weights through code using functions like GetBlendShapeWeight and SetBlendShapeWeight.

You can also check how many blend shapes a Mesh has on it by accessing the blendShapeCount variable along with other useful functions.

Here is an example of code which blends a default shape into two other Blend Shapes over time when attached to a **gameobject** that has 3 or more blend shapes:

```
//Using C#

using UnityEngine;
using System.Collections;

public class BlendShapeExample : MonoBehaviour
{

        int blendShapeCount;
        SkinnedMeshRenderer skinnedMeshRenderer;
        Mesh skinnedMesh;
        float blendOne = 0f;
        float blendTwo = 0f;
        float blendSpeed = 1f;
        bool blendOneFinished = false;

        void Awake ()
        {
```

```
        skinnedMeshRenderer = GetComponent<SkinnedMeshRenderer> ();
        skinnedMesh = GetComponent<SkinnedMeshRenderer> ().sharedMesh;
    }

    void Start ()
    {
        blendShapeCount = skinnedMesh.blendShapeCount;
    }

    void Update ()
    {
        if (blendShapeCount > 2) {

                if (blendOne < 100f) {
                    skinnedMeshRenderer.SetBlendShapeWeight (0, blendOne);
                    blendOne += blendSpeed;
                } else {
                    blendOneFinished = true;
                }

                if (blendOneFinished == true && blendTwo < 100f) {
                    skinnedMeshRenderer.SetBlendShapeWeight (1, blendTwo);
                    blendTwo += blendSpeed;
                }

        }
    }
}
```

Animator Override Controllers

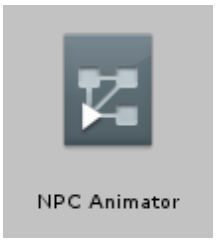
Leave feedback

The **Animator Override Controller** is a type of **asset** which allows you to extend an existing **Animator Controller**, replacing the specific animations used but otherwise retaining the original's structure, parameters and logic.

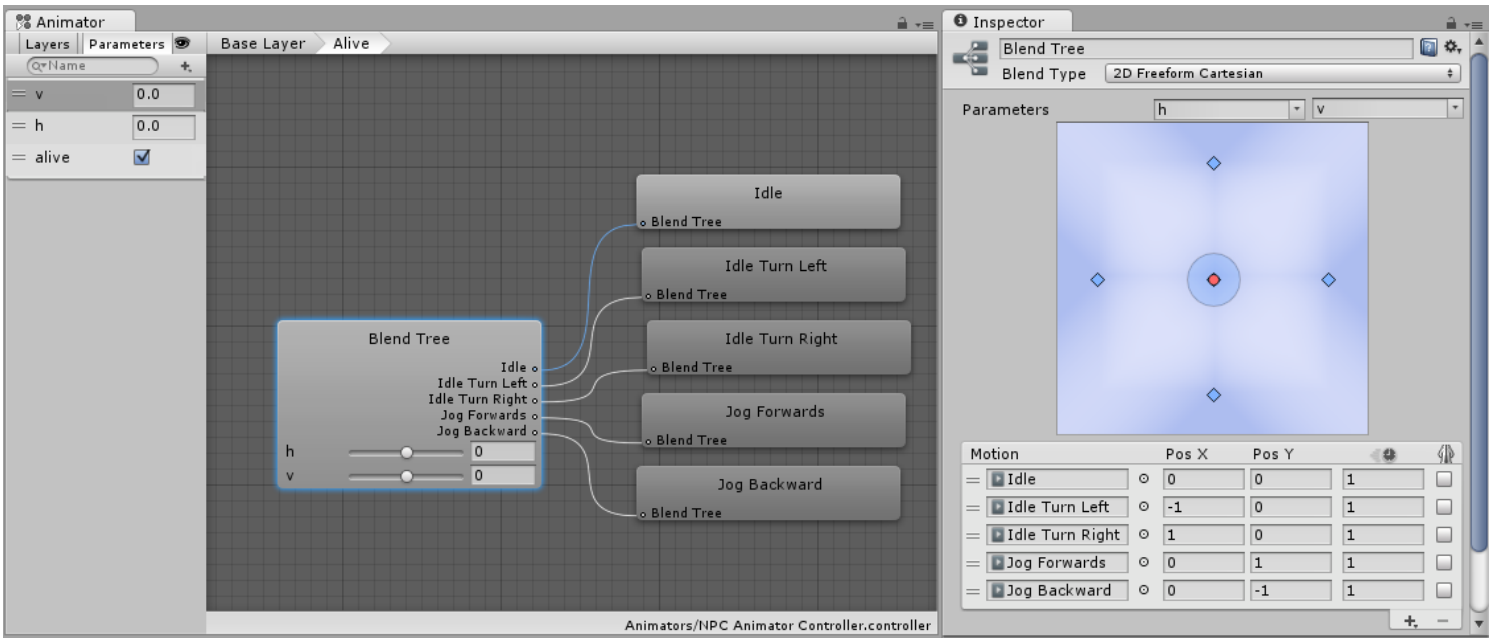
This allows you to create multiple variants of the same basic **state machine**, but with each using different sets of animations. For example, your game may have a variety of NPC types living in the world, but each type (goblin, ogre, elf, etc) has their own unique animations for walking, idling, sitting, **etc**.

By creating one "base" Animator Controller containing the logic for all NPC types, you could then create an override for each type and drop in their respective animation files.

To demonstrate, here's a typical Animator Controller asset:



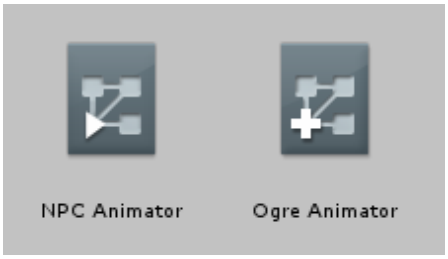
This represents an Animator Controller containing a simple state machine with a blend tree controlling animations in four directions, plus an idle animation, looking like this:



To extend this general NPC state machine to use unique animations which just apply to - say - an ogre-type character, you can create an Animator Override Controller and drop in the Ogre's **animation clips** as replacements to the original **animation clips**. The Ogre may have a different way of idling and moving around, perhaps with slower, heavier and more muscular motion. However, using an Animator Override Controller, the basic logic for how to transistion and blend between movement states can be shared between different characters with different sets of animation, reducing the work required building and modifying state machines themselves.

To create a new Animator Override Controller, use the **Assets -> Create** menu, or the **Create button** in the Project view, and select Animator Override Controller.

The Animator Override Controller has a very similar icon to the Animator Controller, except that it has a “plus” sign rather than a “play” sign in the corner of the icon:

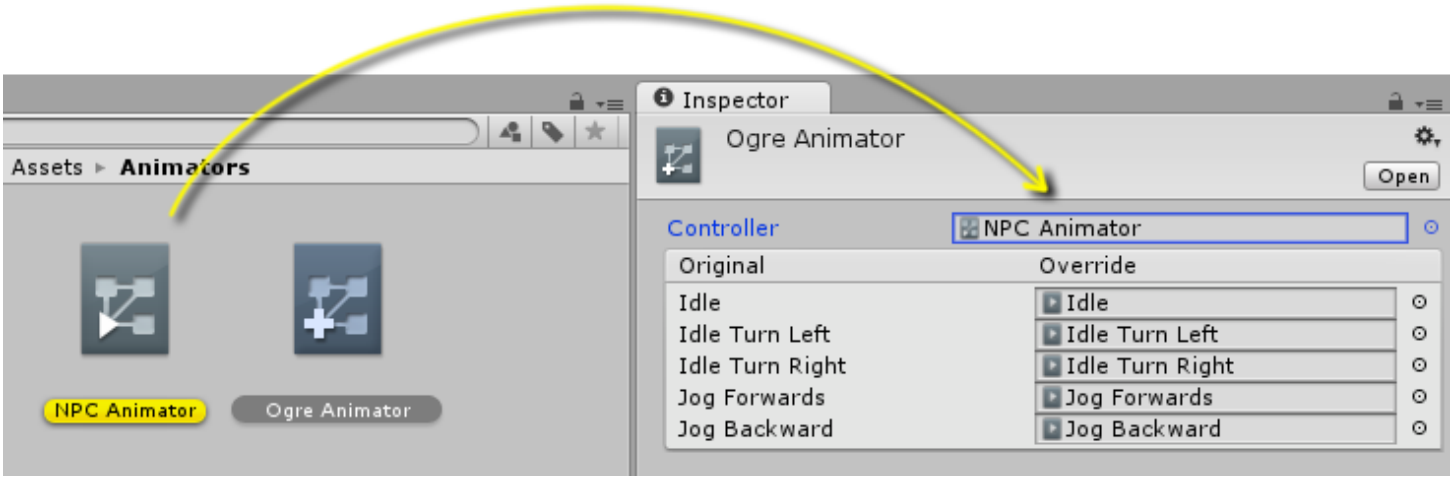


Comparing icons: The Animator Controller and the Animator Override Controller assets side-by-side

When you select the new Animator Override Controller in the **inspector**, it will initially be unassigned, and will look like this:

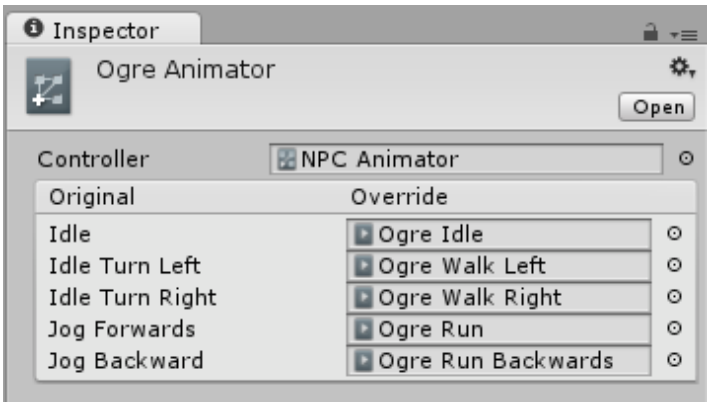
An Animator Override Controller with no Animator Controller assigned

To begin using the Override Controller, you need to assign the original controller asset to the new Override Controller in the inspector. Once this is done, all the animations used in the original controller will show up as a list in the inspector of the override controller:



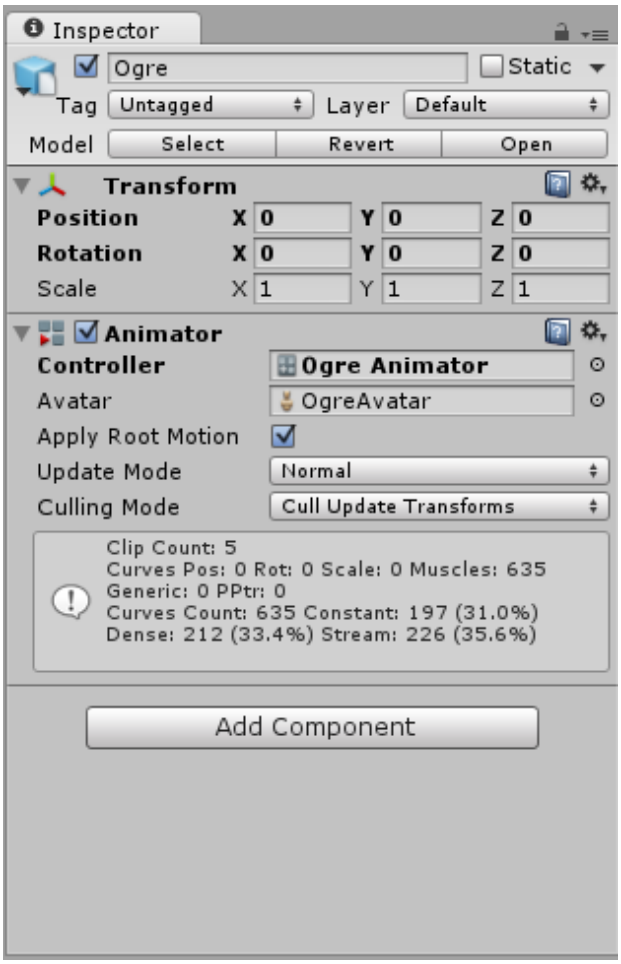
Dragging an existing controller into the Animator Override Controller’s inspector

You can then assign new animation clips to override the original’s clips. In this example, all the clips have been overridden with the “Ogre” versions of the animation.



The Override Controller with new clips assigned

This Override Controller can now be used in an animator **component** on the Ogre character's Game Object just as if it was an Animator Controller. It will use the same logic as the original Animator Controller, but play the new animations assigned instead of the originals.



The Override Controller in use on a Game Object, in the Animator Component

Retargeting of Humanoid animations

Leave feedback

One of the most powerful features of Mecanim is **retargeting** of **humanoid animations**. This means that with relative ease, you can apply the same set of animations to various character models. **Retargeting** is only possible for humanoid models, where an **Avatar** has been configured, because this gives us a correspondence between the models' bone structure.

Recommended Hierarchy structure

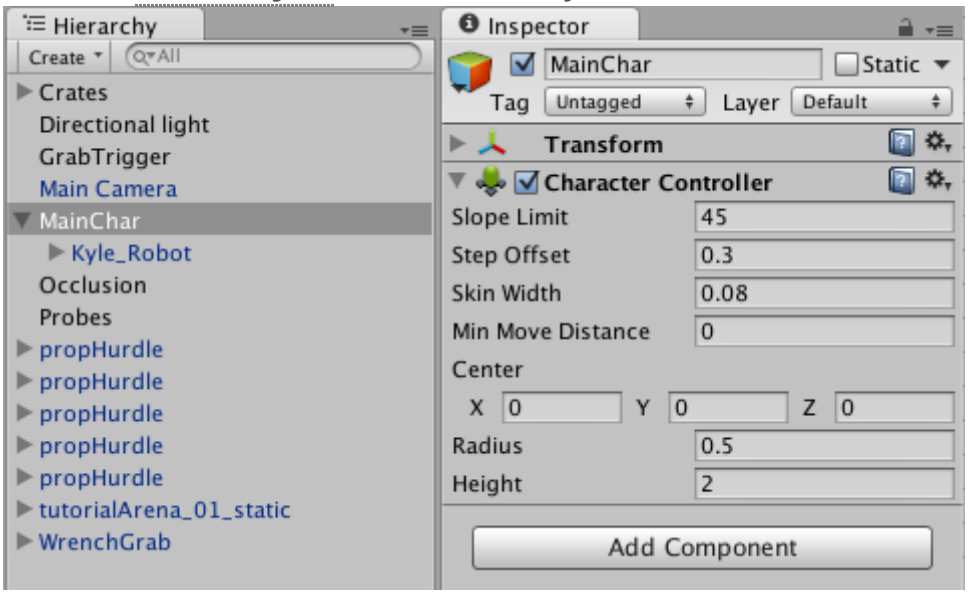
When working with Mecanim animations, you can expect your **scene** to contain the following elements:-

The Imported character model, which has an Avatar on it.
The Animator **Component**, referencing an **Animator Controller asset**.
A set of **animation clips**, referenced from the Animator Controller.
Scripts for the character.
Character-related components, such as the **Character Controller**.
Your project should also contain another character model with a valid Avatar.

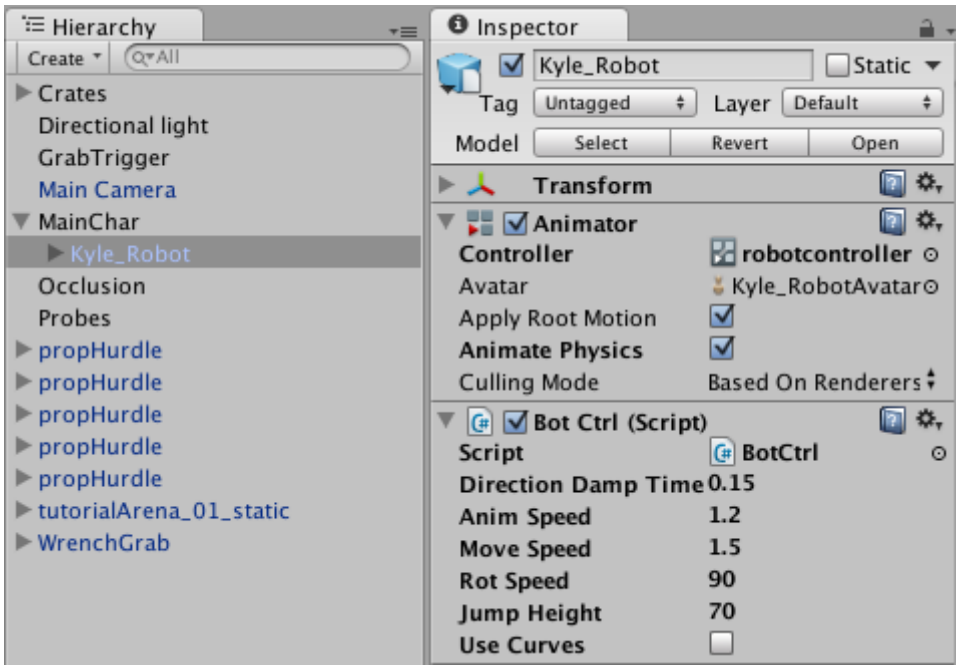
If in doubt about the terminology, consult the Animation Glossary

The recommended setup is to:

Create a **GameObject** in the Hierarchy that contains Character-related components



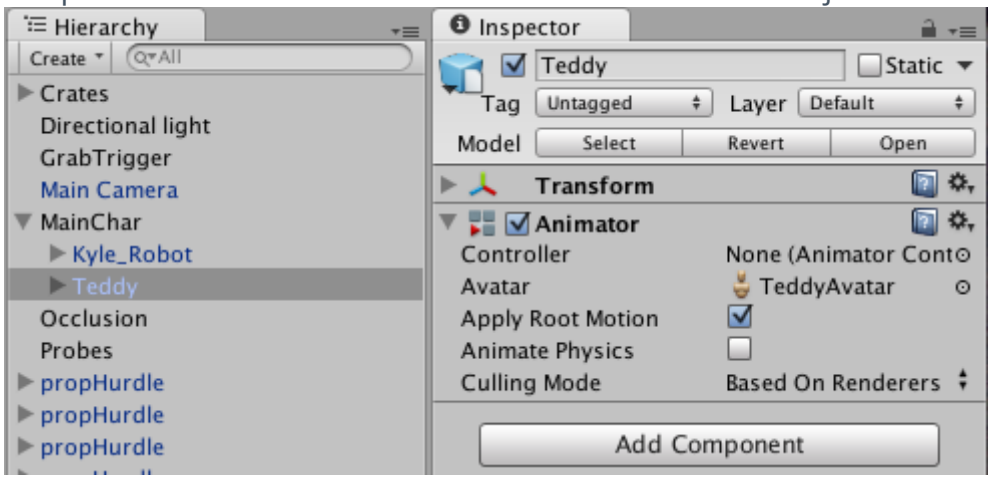
Put the model as a child of the GameObject, together with the Animator component



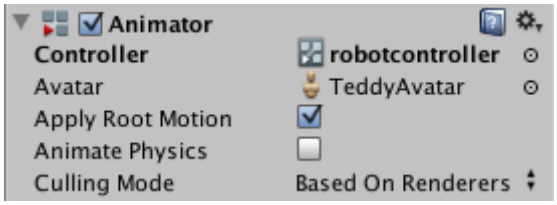
Make sure scripts referencing the Animator are looking for the animator in the children instead of the root; use GetComponentInChildren<Animator>() instead of GetComponent<Animator>().

Then in order to reuse the same animations on another model, you need to:

Disable the original model
Drop in the desired model as another child of GameObject



Make sure the Animator Controller property for the new model is referencing the same controller asset



Tweak the character controller, the transform, and other properties on the top-level GameObject, to make sure that the animations work smoothly with the new model.

You're done!

Performance and optimization

Leave feedback

This page contains some tips to help you obtain the best performance from Unity's animation system, covering character setup, the animation system and run-time optimizations.

Character setup

Number of bones

In some cases you need to create characters with a large number of bones: for example, when you want a lot of customizable attachments. These extra bones increase the size of the build, and may have a relative processing cost for each additional bone. For example, 15 additional bones on a rig that already has 30 bones takes 50% longer to solve in **Generic** mode. Note that you can have additional bones for Generic and Humanoid types. When you have no animations playing using the additional bones, the processing cost should be negligible. This cost is even lower if their attachments are non-existent or hidden.

Multiple skinned Meshes

Combine skinned meshes whenever possible. Splitting a character into two Skinned Mesh Renderers reduces performance. It's better if your character has just one Material, but there are some cases when you might require more than one.

Animation system

Controllers

The Animator doesn't spend time processing when a Controller is not set to it.

Simple animation

Playing a single Animation Clip with no blending can make Unity slower than the legacy animation system. The old system is very direct, sampling the curve and directly writing into the transform. Unity's current animation system has temporary buffers it uses for blending, and there is additional copying of the sampled curve and other data. The current system layout is optimized for animation blending and more complex setups.

Scale curves

Animating scale curves is more expensive than animating translation and rotation curves. To improve performance, avoid scale animations.

Note: This does not apply to constant curves (curves that have the same value for the length of the animation clip). Constant curves are optimized, and are less expensive that normal curves. Constant curves that have the same values as the default **scene** values do not write to the **scene** every frame.

Layers

Most of the time Unity is evaluating animations, and keeps the overhead for animation layers and Animation State Machines to the minimum. The cost of adding another layer to the animator, synchronized or not, depends on what animations and blend trees are played by the layer. When the weight of the layer is zero, Unity skips the layer update.

Humanoid vs. Generic animation types

These are tips to help you decide between these types:

When importing **Humanoid animation** use an **Avatar Mask** (class-AvatarMask) to remove IK Goals or finger animation if you don't need them.
When you use Generic, using **root motion** is more expensive than not using it. If your animations don't use **root motion**, make sure that you have not specified a root bone.

Scene-level optimization

There are many optimizations that can be made, some useful tips include:

Use hashes instead of strings to query the Animator.
Implement a small AI Layer to control the Animator. You can make it provide simple callbacks for OnStateChange, OnTransitionBegin, and other events.
Use State Tags to easily match your AI **state machine** to the Unity **state machine**.
Use additional curves to simulate events.
Use additional curves to mark up your animations; for example, in conjunction with target matching.

Runtime Optimizations

Visibility and updates

Always optimize animations by setting the animators's Culling Mode to **Based on Renderers**, and disable the skinned mesh renderer's **Update When Offscreen** property. This saves Unity from updating animations when the character is not visible.

2018–04–25 Page amended with limited editorial review

2017–05–16 Page amended with no editorial review

Animation Reference

Leave feedback

For a detailed explanation of the Unity Animation System, please see the Animation System Overview introduction.

For information about importing Animation from your Model, see Model import workflows or the Animation tab reference page.

2018–04–25 Page amended with limited editorial review

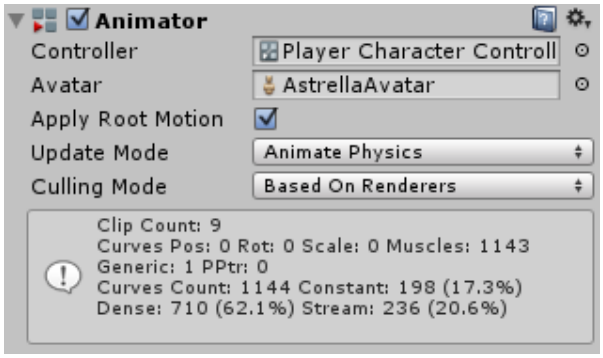
Animator Component

Leave feedback

SWITCH TO SCRIPTING

The Animator **component** is used to assign animation to a **GameObject** in your **scene**. The Animator **component** requires a reference to an Animator Controller which defines which **animation clips** to use, and controls when and how to blend and transition between them.

If the GameObject is a humanoid character with an Avatar definition, the Avatar should also be assigned in this component, as seen here:



The Animator component with a controller and avatar assigned.

This diagram shows how the various **assets** (Animation Clips, an Animator Controller, and an Avatar) are all brought together in an **Animator Component** on a Game Object:

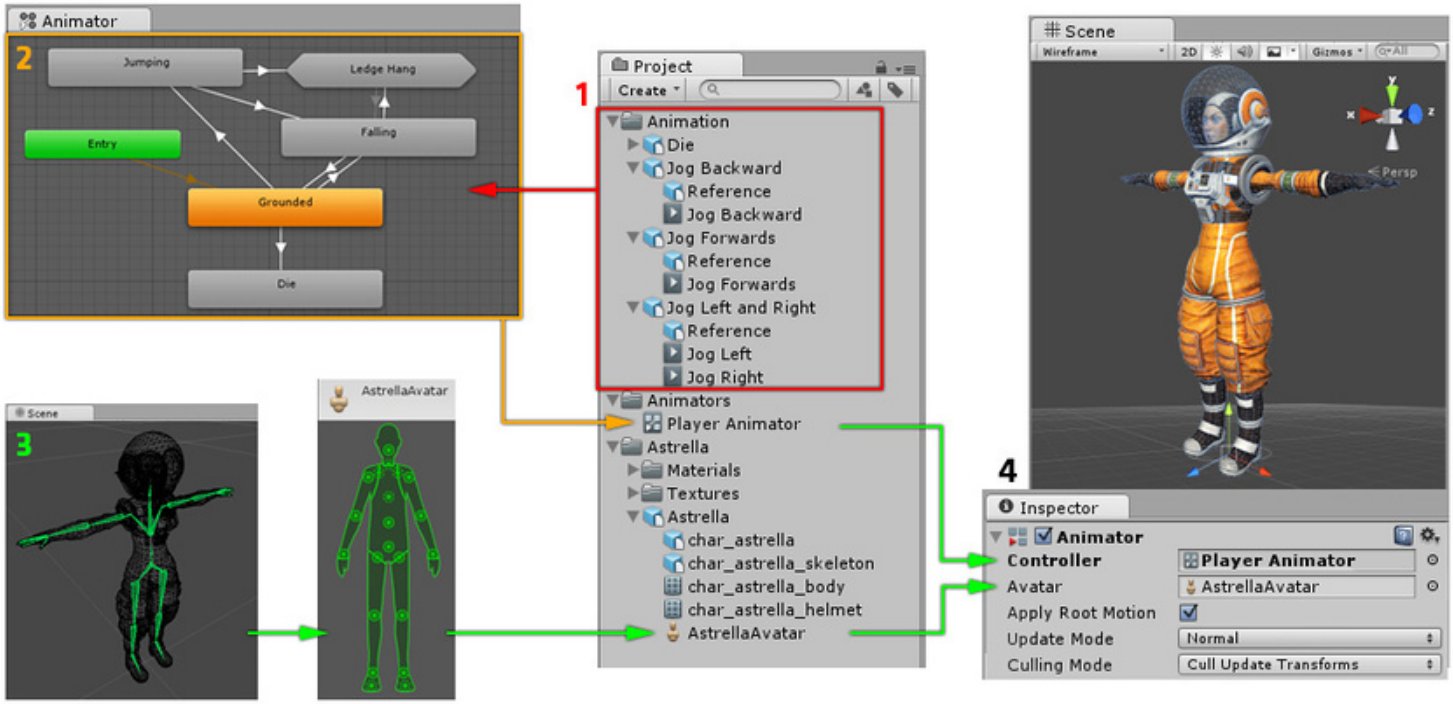


Diagram showing how the various parts of the animation system connect together

See Also State Machines, Blend Trees, Avatar, Animator Controller

Properties

Property:	*Function:*
Controller	The animator controller attached to this character.

Property:	*Function:*
Avatar	The Avatar for this character. (If the Animator is being used to animate a humanoid character)
Apply Root Motion	Should we control the character's position and rotation from the animation itself or from script.
Update Mode	This allows you to select when the Animator updates, and which timescale it should use.
Normal	The animator is updated in-sync with the Update call, and the animator's speed matches the current timescale. If the timescale is slowed, animations will slow down to match.
Animate Physics	The animator is updated in-sync with the FixedUpdate call (i.e. in lock-step with the physics system). You should use this mode if you are animating the motion of objects with physics interactions, such as characters which can push **rigidbody** objects around.
Unscaled Time	The animator is updated in-sync with the Update call, but the animator's speed ignores the current timescale and animates at 100% speed regardless. This is useful for animating a GUI system at normal speed while using modified timescales for special effects or to pause gameplay.
Culling Mode	Culling mode you can choose for animations.
Always Animate	Always animate, don't do culling even when offscreen.
Cull Update Transforms	Retarget, IK and write of Transforms are disabled when renderers are not visible.
Cull Completely	Animation is completely disabled when renderers are not visible.

Animation curve information

The information box at the bottom of the Animator component provides you with a breakdown of the data being used in all the clips used by the Animator Controller.

An animation clip contains data in the form of "curves", which represent how a value changes over time. These curves may describe the position or rotation of an object, the flex of a muscle in the **humanoid animation** system, or other animated values within the clip such as a changing material colour.

This table explains what each item of data represents:

Label	*Description*
Clip Count	The total number of animation clips used by the animator controller assigned to this Animator.
Curves (Pos, Rot & Scale)	The total number of curves used to animate the position, rotation or scale of objects. These are for animated objects that are not part of a standard humanoid rig. When animating a humanoid avatar, these curves would show up a count for extra non-muscle bones such as a tail, flowing cloth or a dangling pendant. If you have a humanoid animation and you notice unexpected non-muscle **animation curves**, you mave have unnecessary **animation curves** in your animation files.
Muscles	The number of muscle animation curves used for humanoid animation by this Animator. These are the curves used to animate the standard humanoid avatar muscles. As well as the standard muscle movements for all the humanoid bones in Unity's standard avatar, this also includes two "muscle curves" which store the **root motion** position and rotation animation.

Label	*Description*
Generic	The number of numeric (float) curves used by the animator to animate other properties such as material colour.
PPtr	The total count of **sprite** animation curves (used by Unity's 2d system)
Curves Count	The total combined number of animation curves
Constant	The number of animation curves that are optimised as constant (unchanging) values. Unity selects this automatically if your animation files contain curves with unchanging values.
Dense	The number of animation curves that are optimised using the "dense" method of storing data (discrete values which are interpolated between linearly). This method uses less significantly less memory than the "stream" method.
Stream	The number of animation curves using the "stream" method of storing data (values with time and tangent data for curved interpolation). This data occupies significantly more memory than the "dense" method.

If your animation clips are imported with "Anim **Compression**" set to "Optimal" in the Animation import reference, Unity will use a heuristic algorithm to determine whether it is best to use the dense or stream method to store the data for each curve.

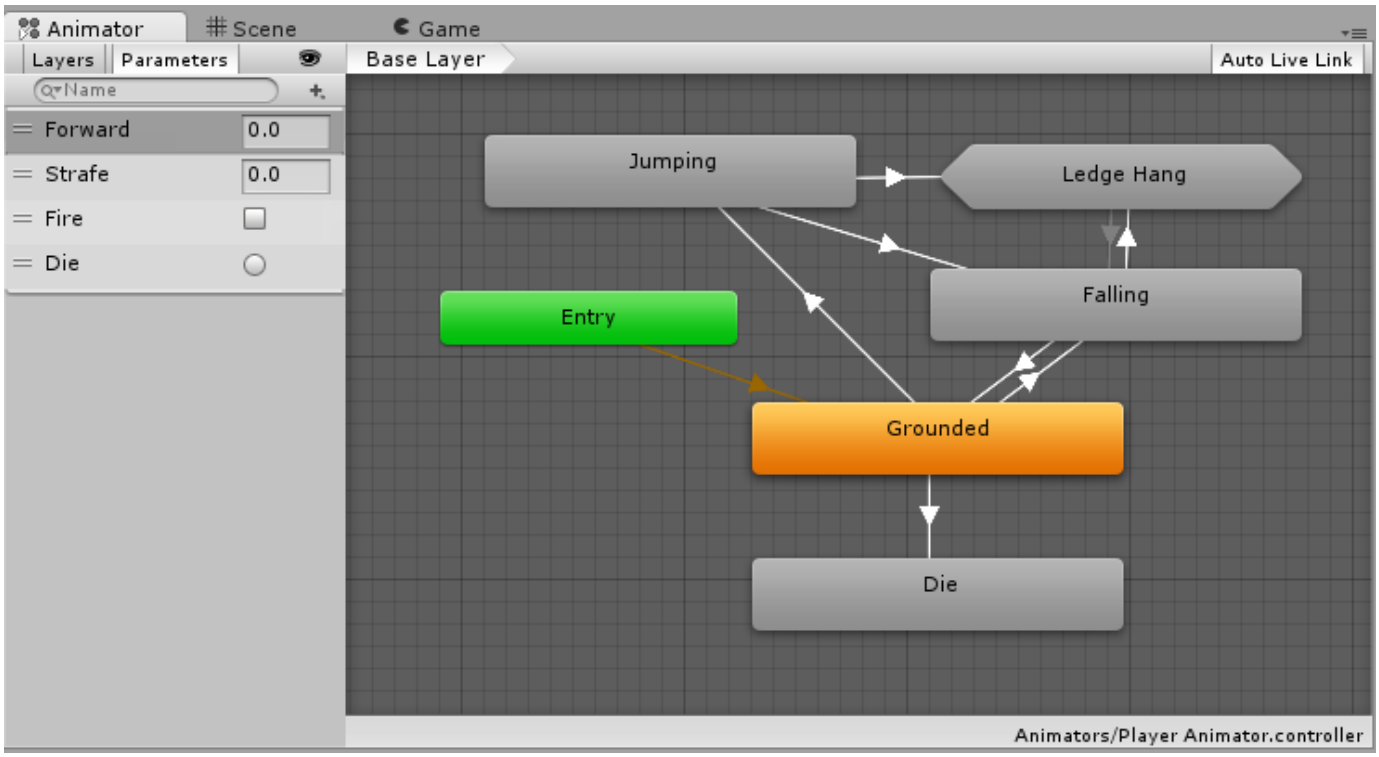
2018–04–25 Page amended with limited editorial review

Animator Controller

Leave feedback

An **Animator Controller** allows you to arrange and maintain a set of Animation Clips and associated **Animation Transitions** for a character or object. In most cases it is normal to have multiple animations and switch between them when certain game conditions occur. For example, you could switch from a walk **Animation Clip** to a jump **Animation Clip** whenever the spacebar is pressed. However even if you only have a single **Animation Clip** you still need to place it into an **Animator Controller** to use it on a **GameObject**.

The Animator Controller has references to the Animation clips used within it, and manages the various Animation Clips and the Transitions between them using a **State Machine**, which could be thought of as a flow-chart of Animation Clips and Transitions, or a simple program written in a visual programming language within Unity. More information about state machines can be found here.



A simple Animator Controller

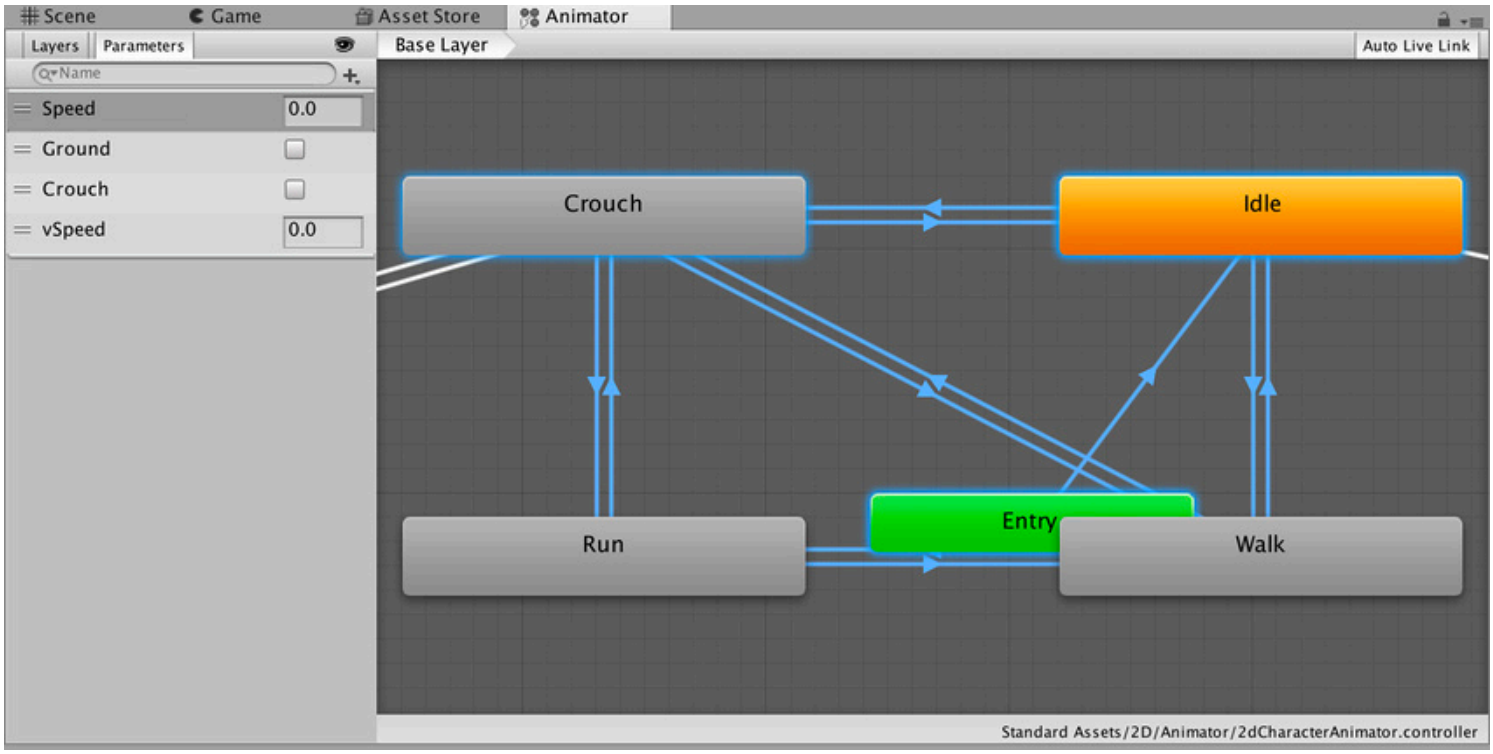
Unity automatically creates an Animator Controller when you begin animating a GameObject using the Animation Window, or when you attach an Animation Clip to a GameObject.

To manually create an Animator Controller, right click the Project window and click **Create > Animator Controller**.

Navigation

Use the scroll wheel on your mouse, or equivalent, to zoom in and out of the Animator Controller window.

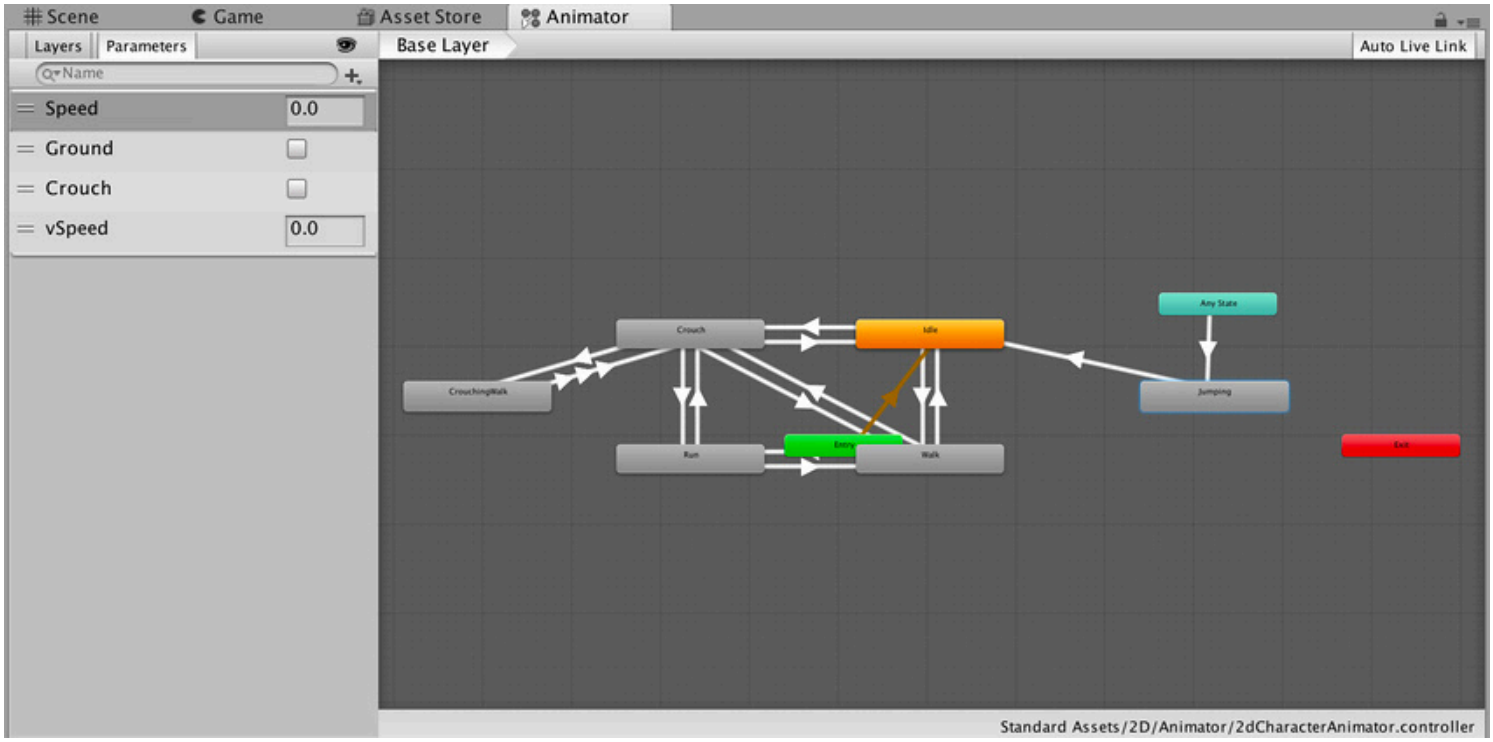
To focus on an item in the Animator Controller window, select one or multiple states (click or drag a selection box around the states you wish to select), then press the F key to zoom in on the selection.



Focus on selected states

Press the A key to fit all of the animation states into the Animator Controller view.

Unity preserves your selection. Press the A and F keys to switch between your selected animation states and the entire Animator Controller.

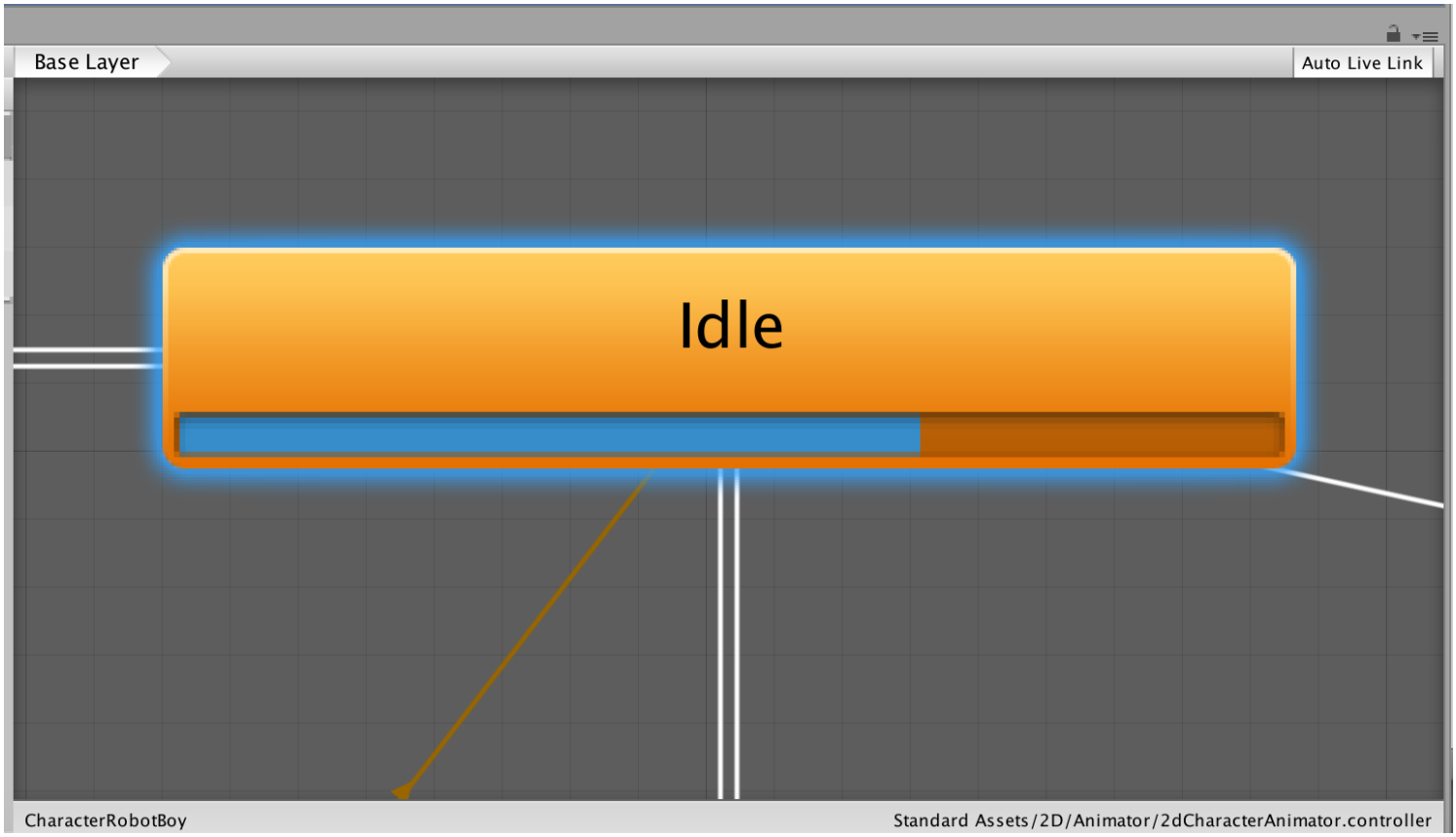


Unity automatically fits all states in the Animator Controller view when the A key is pressed

During Play Mode, the Animator pans the view so that the current state being played is always visible. The Animator Controller respects the independent zoom factors of the Base Layer and Sub-State Machine, and the window pans automatically to ensure visibility of the active state or states.

To modify the zoom during Play Mode, follow these steps:

Enable **Auto Live Link** in the Animator Controller window
Click the Play button to enter Play Mode
Click Pause
In the Animator Controller, select the state or states you want to zoom into
Press the F key to zoom into the selection
Click the Play button again to resume Play Mode
Note that the Animator Controller pans to each state when it activates.



The Animator pans to the active state

2017–11–21 Page published with limited editorial review

Animator zoom added in 2017.3

Creating an AnimatorController

Leave feedback

Animator Controller

You can view and set up character behavior from the **Animator Controller** view (Menu: **Window** > **Animation** > **Animator**).

The various ways an Animator Controller can be created:

From the **Project View** by selecting '**Create > Animator Controller**'.

By right-clicking in the Project View and selecting '**Create > Animator Controller**'.

From the Assets menu by selecting '**Assets > Create > Animator Controller**'.

This creates a `.controller` **asset** on disk. In the **Project Browser** window the icon will look like:

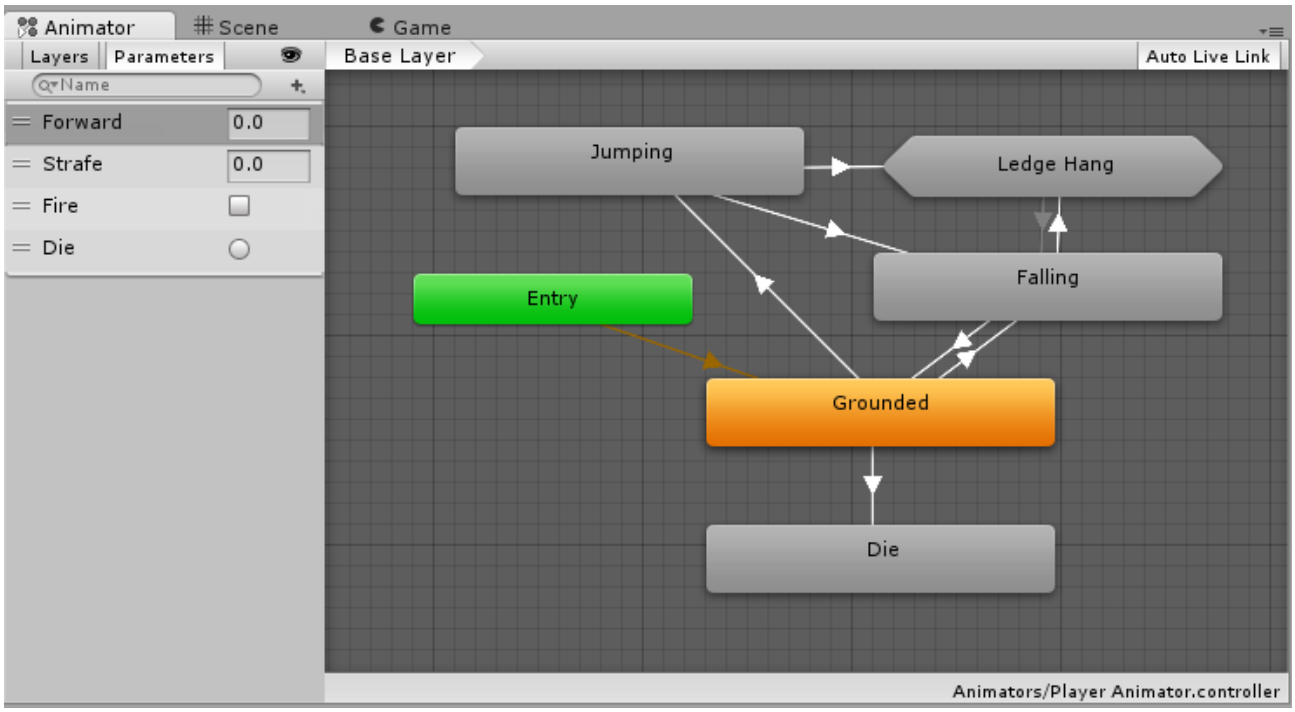
Animator Controller asset on disk

Animator Window

After the **state machine** setup has been made, you can drop the controller onto the Animator **component** of any character with an **Avatar** in the **Hierarchy View**.

The Animator Controller window contains:

The **Animation Layer Widget** (top-left corner, see Animation Layers)
The **Event Parameters Widget** (top-left, see Animation Parameters)
The visualization of the State Machine itself.



The Animator Controller Window

Note that the Animator Controller Window will always display the state machine from the most recently selected `.controller` asset, regardless of what **scene** is currently loaded.

Animation States

Leave feedback

Animation States are the basic building blocks of an **Animation State Machine**. Each state contains an individual animation sequence (or blend tree) which will play while the character is in that state. When an event in the game triggers a state transition, the character will be left in a new state whose animation sequence will then take over.

When you select a state in the **Animator Controller**, you will see the properties for that state in the **inspector**:-

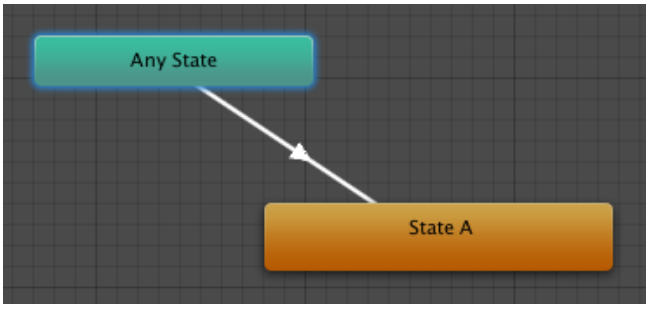
Property:	*Function:*
Speed	The default speed of the animation
Motion	The **animation clip** assigned to this state
Foot IK	Should Foot IK be respected for this state. Applicable to **humanoid animations**.
Write Defaults	Whether or not the AnimatorStates writes back the default values for properties that are not animated by its Motion.
Mirror	Should the state be mirrored. This is only applicable to humanoid animations.
Transitions	The list of transitions originating from this state

The default state, displayed in brown, is the state that the machine will be in when it is first activated. You can change the default state, if necessary, by right-clicking on another state and selecting **Set As Default** from the context menu. The *solo* and *mute* checkboxes on each transition are used to control the behaviour of **animation previews** - see this page for further details.

A new state can be added by right-clicking on an empty space in the **Animator Controller Window** and selecting **Create State->Empty** from the context menu. Alternatively, you can drag an animation into the Animator Controller Window to create a state containing that animation. (Note that you can only drag Mecanim animations into the Controller - non-Mecanim animations will be rejected.) States can also contain Blend Trees.

Any State

Any State is a special state which is always present. It exists for the situation where you want to go to a specific state regardless of which state you are currently in. This is a shorthand way of adding the same outward transition to all states in your machine. Note that the special meaning of **Any State** implies that it cannot be the end point of a transition (ie, jumping to “any state” cannot be used as a way to pick a random state to enter next).

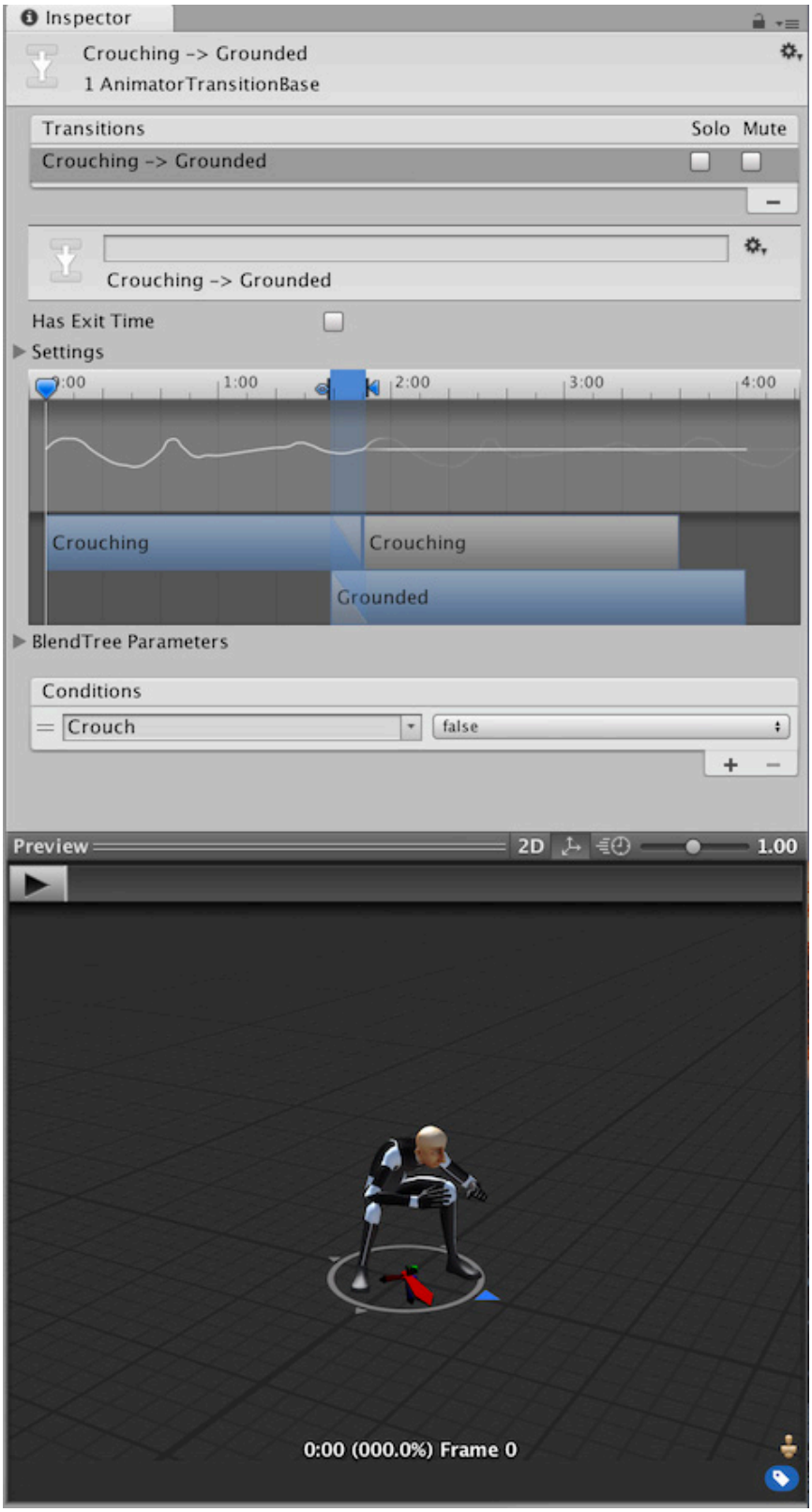


Animation transitions

Leave feedback

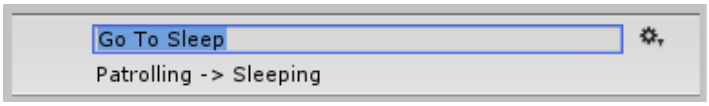
Animation transitions allow the state machine to switch or blend from one animation state to another. Transitions define not only how long the blend between states should take, but also under what conditions they should activate. You can set up a transition to occur only when certain conditions are true. To set up these conditions, specify values of parameters in the **Animator Controller**.

For example, your character might have a “patrolling” state and a “sleeping” state. You could set the transition between patrolling and sleeping to occur only when an “alertness” parameter value drops below a certain level.

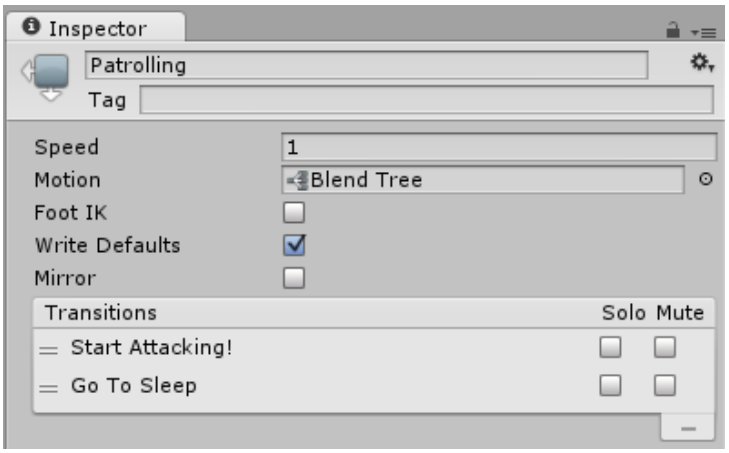


An example of a transition as viewed in the inspector.

To give transitions a name, type it into the field as shown below:



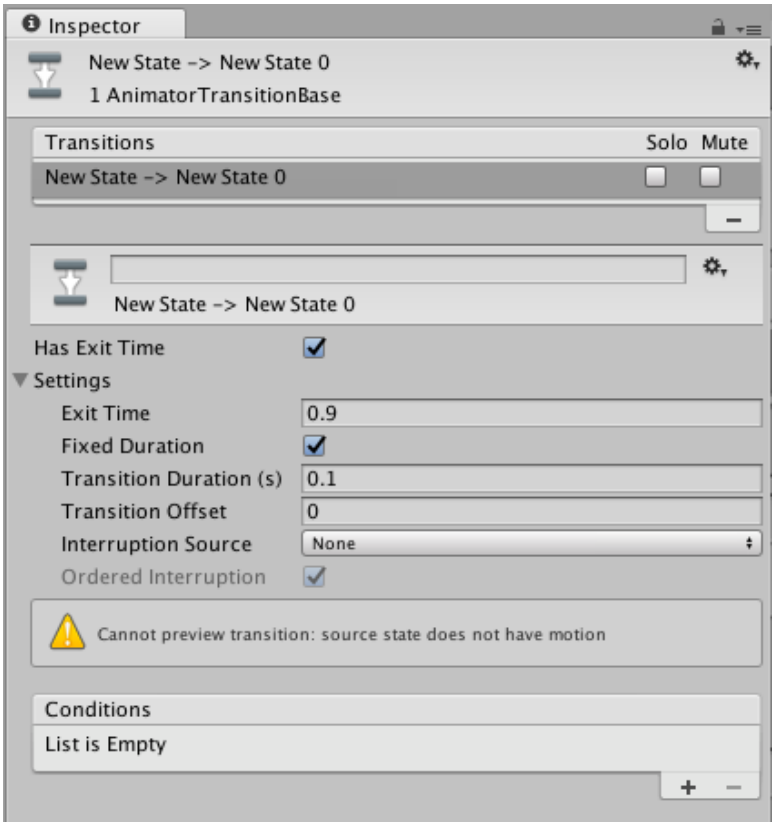
The **Inspector** window of a state shows the transitions the state uses as shown below:



There can be only one transition active at any given time. However, the currently active transition can be interrupted by another transition if you have configured the settings to allow it (see Transition Interruption below).

Transition properties

To view properties for a transition, click on the transition line connecting two states in the **Animator window**. The properties appear in the Inspector window.



Use the following properties to adjust the transition and how it blends between the current and next state.

Property	Function
Has Exit Time	**Exit Time** is a special transition that doesn't rely on a parameter. Instead, it relies on the normalized time of the state. Check to make the transition happen at the specific time specified in **Exit Time**.
Settings	Fold-out menu containing detailed transition settings as below.

Property	Function
Exit Time	If **Has Exit Time** is checked, this value represents the exact time at which the transition can take effect. This is represented in normalized time (for example, an exit time of 0.75 means that on the first frame where 75% of the animation has played, the **Exit Time** condition is true). On the next frame, the condition is false. For looped animations, transitions with exit times smaller than 1 are evaluated every loop, so you can use this to time your transition with the proper timing in the animation every loop. Transitions with an **Exit Time** greater than 1 are evaluated only once, so they can be used to exit at a specific time after a fixed number of loops. For example, a transition with an exit time of 3.5 are evaluated once, after three and a half loops.
Fixed Duration	If the **Fixed Duration** box is checked, the transition time is interpreted in seconds. If the **Fixed Duration** box is not checked, the transition time is interpreted as a fraction of the normalized time of the source state.
Transition Duration	The duration of the transition, in normalized time or seconds depending on the **Fixed Duration** mode, relative to the current state's duration. This is visualized in the transition graph as the portion between the two blue markers.
Transition Offset	The offset of the time to begin playing in the destination state which is transitioned to. For example, a value of 0.5 means the target state begins playing at 50% of the way through its own **timeline**.
Interruption Source	Use this to control the circumstances under which this transition may be interrupted (see Transition interruption below).
Ordered Interruption	Determines whether the current transition can be interrupted by other transitions independently of their order (see Transition interruption below).
Conditions	A transition can have a single condition, multiple conditions, or no conditions at all. If your transition has no conditions, the Unity Editor only considers the **Exit Time**, and the transition occurs when the exit time is reached. If your transition has one or more conditions, the conditions must all be met before the transition is triggered. A condition consists of: - An event parameter (the value considered in the condition). - A conditional predicate (if needed,for example, 'less than' or 'greater than' for floats). - A parameter value (if needed). If you have **Has Exit Time** selected for the transition and have one or more conditions, note that the Unity Editor considers whether the conditions are true after the **Exit Time**. This allows you to ensure that your transition occurs during a certain portion of the animation.

Transition interruption

Use the **Interruption Source** and **Ordered Interruption** properties to control how your transition can be interrupted.

The interruption order works, conceptually, as if transitions are queued and then parsed for a valid transition from the first transition inserted to the last.

Interruption Source property

The transitions in AnyState are always added first in the queue, then other transitions are queued depending on the value of **Interruption Source**:

Value	Function
None	Don't add any more transitions.
Current State	Queue the transitions from the current state.
Next State	Queue the transitions from the next state.
Current State then Next State	Queue the transitions from the current state, then queue the ones from the next state.
Next State then Current State	Queue the transitions from the next state, then queue the ones from the current state.

Note: This means that even with the **Interruption Source** set to **None**, transitions can be interrupted by one of the AnyState transitions.

Ordered Interruption property

The property **Ordered Interruption** changes how the queue is parsed.

Depending on its value, parsing the queue ends at a different moment as listed below.

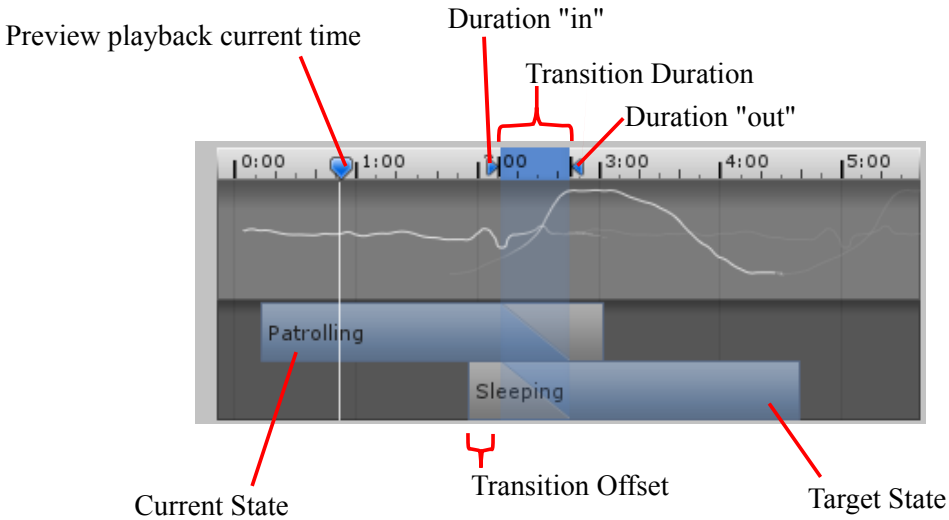
Value	Ends when
Checked	A valid transition or the current transition has been found.
Unchecked	A valid transition has been found.

Only an AnyState transition can be interrupted by itself.

To learn more about transition interruptions, see the Unity blog post State Machine Transition Interruptions.

Transition graph

To manually adjust the settings listed above, you can either enter numbers directly into the fields or use the transition graph. The transition graph modifies the values above when the visual elements are manipulated.



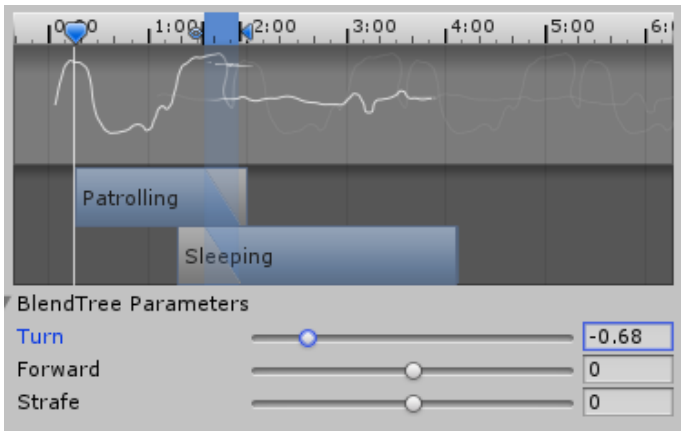
The Transition settings and graph as shown in the Inspector

Change the transition properties in the graph view using the following directions::

Drag the **Duration "out"** marker to change the **Duration** of the transition.
Drag the **Duration "in"** marker to change the duration of the transition and the **Exit Time**.
Drag the target state to adjust the **Transition Offset**.
Drag the preview playback marker to scrub through the animation blend in the preview window at the bottom of the Inspector.

Transitions between Blend Tree states

If either the current or next state belonging to this transition is a Blend Tree state, the Blend Tree parameters appear in the Inspector. Adjust these values to preview how your transition would look with the Blend Tree values set to different configurations. If your Blend Tree contains clips of differing lengths, you should test what your transition looks like when showing both the short clip and the long clip. Adjusting these values does not affect how the transition behaves at runtime; they are solely for helping you preview how the transition could look in different situations.



The Blend Tree parameter preview controls, visible when either your current or next state is a Blend Tree state.

Conditions

A transition can have a single condition, multiple conditions, or no conditions at all. If your transition has no conditions, the Unity Editor only considers the **Exit Time**, and the transition occurs when the exit time is reached. If your transition has one or more conditions, the conditions must all be met before the transition is triggered.

A condition consists of:

An event parameter, the value of which is considered in the condition.
A conditional predicate, if needed (for example, less or greater for floats).
A parameter value, if needed.
If **Has Exit Time** is enabled for the transition and has one or more conditions, these conditions are only checked after the exit time of the state. This allows you to ensure that your transition only occurs during a certain portion of the animation.

Animation FAQ

Leave feedback

General questions

What's "Mecanim"?

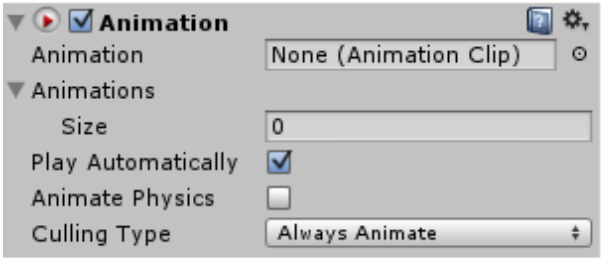
Mecanim was the name of the animation software that we integrated into Unity. Early in the 4.x series of Unity, its abilities were tied specifically to humanoid character animation and it had many features which were uniquely suited for that purpose, and it was separate to our old (now legacy) integrated animation system.

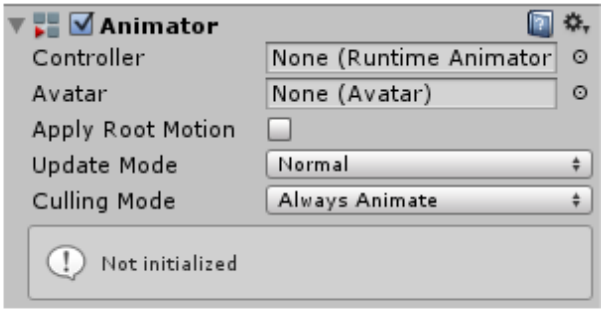
Mecanim integrated **humanoid animation retargeting**, muscle control, and the **state machine** system. The name "Mecanim" comes from the French word "Mec" meaning "Guy". Since Mecanim operated only with humanoid characters, our legacy animation system was still required for animating non-humanoid characters and other keyframe-based animation of **gameobjects** within Unity.

Since then however, we've developed and expanded Mecanim and integrated it with the rest of our animation system so that it can be used for all aspects of animation within your project - so there is a less clear definition where "Mecanim" ends and the rest of the animation system begins. For this reason, you'll still see references in our documentation and throughout our community to "Mecanim" which has now simply come to mean our main animation system.

What's the difference between the Animation component and the Animator component?

The Animation component is an old component used for animation in our legacy animation system. It remains in Unity for backwards compatibility but you should not use it for new projects. Use the up-to-date **Animator component** instead.





LEFT: Old Legacy "Animation" component. RIGHT: Modern "Animator" Component

What's the difference between the Animation window and the Animator window?

The **Animation Window** allows you to create and edit **animation clips** within Unity. You can use it to animate almost every property that you can edit in the **inspector**, from a Game Object's position, a material colour, a light's brightness, a sound's volume, and even arbitrary values in your own **scripts**.

The **Animator Window** allows you to organise your existing animation clip **assets** into a flowchart-like system called a state machine.

Both of these windows are part of our current animation system, and not the legacy system.

So the Animation Component is legacy, but the Animation Window is current?

That's correct.

We are using the legacy animation system for character animations. Should we be using the current animation system (Mecanim) instead?

Generally, yes you should. Our legacy animation system is only included for backward compatibility with old projects, and it has a very limited feature set compared with our current animation system. The only reason you should use it is for legacy projects built using the old system.

Import

Why does my imported mesh have an animator component attached to it?

When Unity detectes that an imported file has animation in its **timeline**, it will add an animation component on import. You can modify this in the asset's import settings by setting the "Animation Type" to *None* in the import settings under the Rig tab. If necessary you can do this with several files at once.

Layers

Does the ordering of the layers matter?

Yes. Layers are evaluated from top to bottom in order. Layers set to *override* will always override the previous layers (based on their mask, if they have a mask).

Should the weight value of the base layer always be set to one or should the weight be zero when using another layer?

The base layer weight is always 1 and override layers will completely override the base layer.

Is there any way to get a variable value from the controller without using the name string?

You can use integers to identify the states and parameters. Use the Animator.StringToHash function to get the integer identifier values. For example:

```
runState = Animator.StringToHash("Base Layer.Run");
animator.SetBool(runState, false);
```

What happens if a state on a Sync layer has a different length compared to the corresponding state in the base layer?

If layers have different lengths then they will become unsynchronized. Enable the Timing option to force the timing of the states on the current layer, on the source layer.

Avatar Masks

Is there a way to create AvatarIKGoals other than LeftFoot, RightFoot, LeftHand, RightHand?

Yes, knee and elbow IK is supported.

Is there a way to define what transforms are part of the Avatar Mask?

Yes, for Generic clips you can define which transform animation is imported or not. For Humanoid clips, all human transforms are always imported and extra transforms can de defined.

Animations curves

How do animations that have Curves blend with those that don't?

When you have an animation with a curve and another animation without a curve, Unity will use the default value of the parameter connected to the curve to do blending. You can set default values for your parameters, so when blending takes place between a State that has a *Curve* Parameter and one that does not have one, it will blend between the curve value and the default parameter value. To set a default value for a Parameter, simply set its value in the Animator Tool window while not in LiveLink.

Playables API

Leave feedback

The **Playables** API provides a way to create tools, effects or other gameplay mechanisms by organizing and evaluating data sources in a tree-like structure known as the PlayableGraph. The PlayableGraph allows you to mix, blend, and modify multiple data sources, and play them through a single output.

The Playables API supports animation, audio and **scripts**. The Playables API also provides the capacity to interact with the animation system and audio system through scripting.

Although the Playables API is currently limited to animation, audio, and scripts, it is a generic API that will eventually be used by video and other systems.

Playable vs Animation

The animation system already has a graph editing tool, it's a **state machine** system that is restricted to playing animation. The Playables API is designed to be more flexible and to support other systems. The Playables API also allows for the creation of graphs not possible with the **state machine**. These graphs represent a flow of data, indicating what each node produces and consumes. In addition, a single graph is not limited to a single system. A single graph may contain nodes for animation, audio, and scripts.

Advantages of using the Playables API

The Playables API allows for dynamic animation blending. This means that objects in the **scenes** could provide their own animations. For example, animations for weapons, chests, and traps could be dynamically added to the PlayableGraph and used for a certain duration.

The Playables API allows you to easily play a single animation without the overhead involved in creating and managing an AnimatorController **asset**.

The Playables API allows users to dynamically create blending graphs and control the blending weights directly frame by frame.

A PlayableGraph can be created at runtime, adding playable node as needed, based on conditions. Instead of having a huge "one-size-fit-all" graph where nodes are enabled and disabled, the PlayableGraph can be tailored to fit the requirements of the current situation.

2017–07–04 Page published with limited editorial review

2017–07–04 New in Unity 2017.1

The PlayableGraph

Leave feedback

The PlayableGraph defines a set of playable outputs that are bound to a **GameObject** or component. The PlayableGraph also defines a set of **playables** and their relationships. Figure 1 provides an example.

The PlayableGraph is responsible for the life cycle of its playables and their outputs. Use the PlayableGraph to create, connect, and destroy playables.

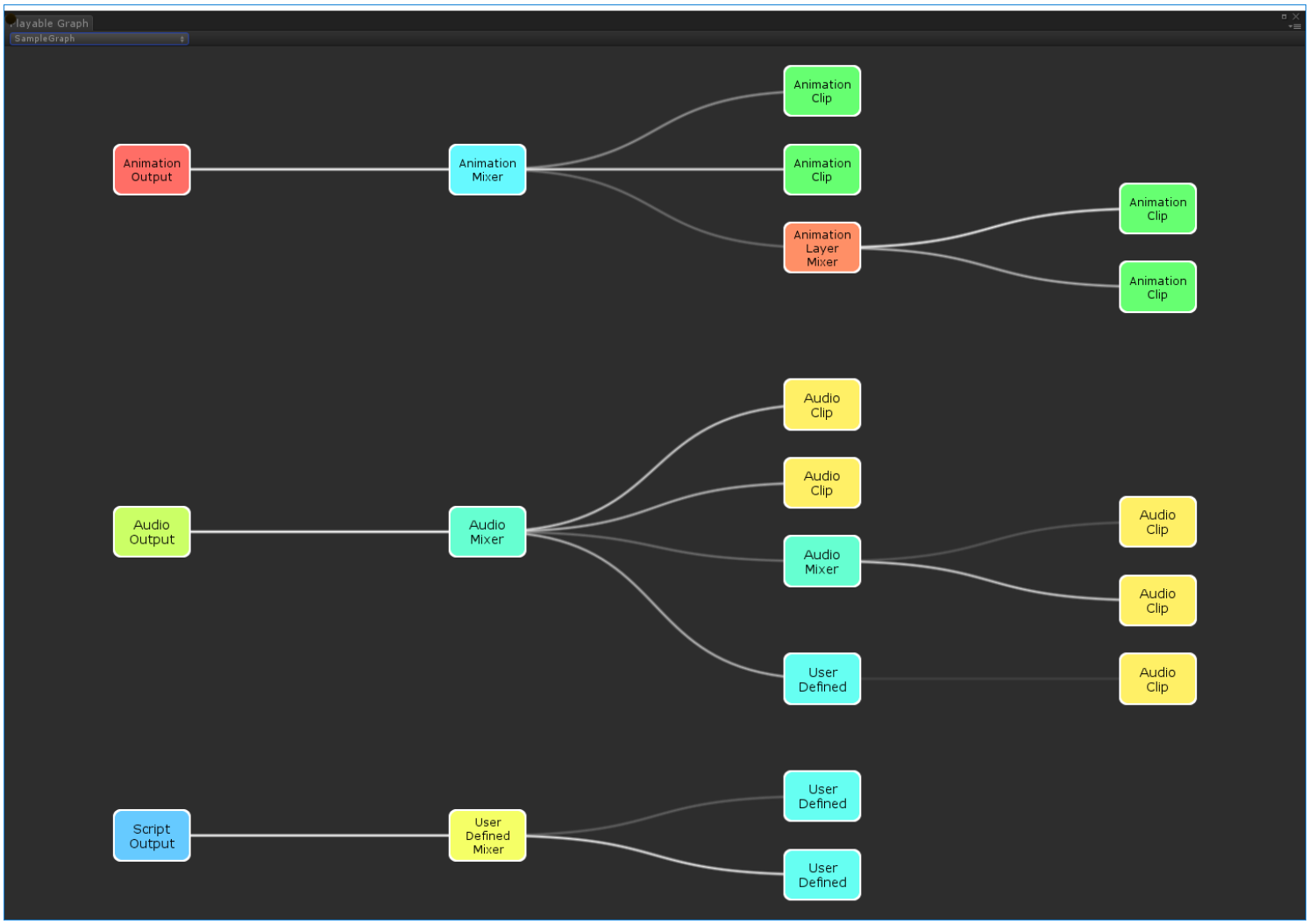


Figure 1: A sample PlayableGraph

In Figure 1, when displaying a PlayableGraph, the term “Playable” is removed from the names of graph nodes to make it more compact. For example, the node named “AnimationClipPlayable” is shown as “AnimationClip.”

	This document uses inheritance nomenclature when describing playables and their outputs. C# inheritance is not used because playable core types and playable output types are implemented as C# structs. However, since these objects mostly behave as if they were using inheritance, the nomenclature is used for brevity and clarity.

A playable is a C# struct that implements the IPlayable interface. It is used to define its relationship with other playables. Likewise, a playable output is a C# struct that implements IPlayableOutput and is used to define the output of a PlayableGraph.

Figure 2 shows the most common core playable types. Figure 3 shows the core playable output types.

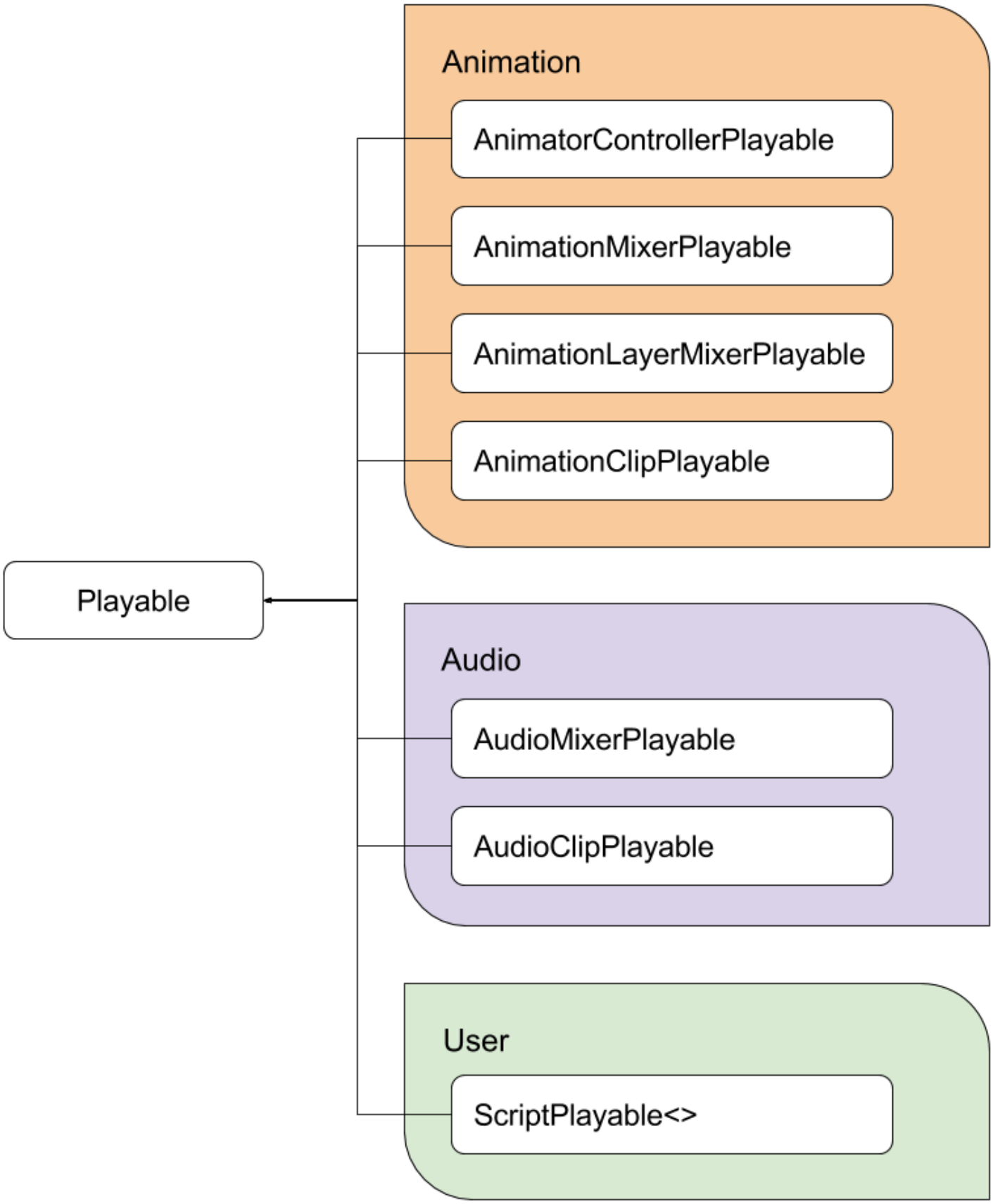


Figure 2: Core playable types

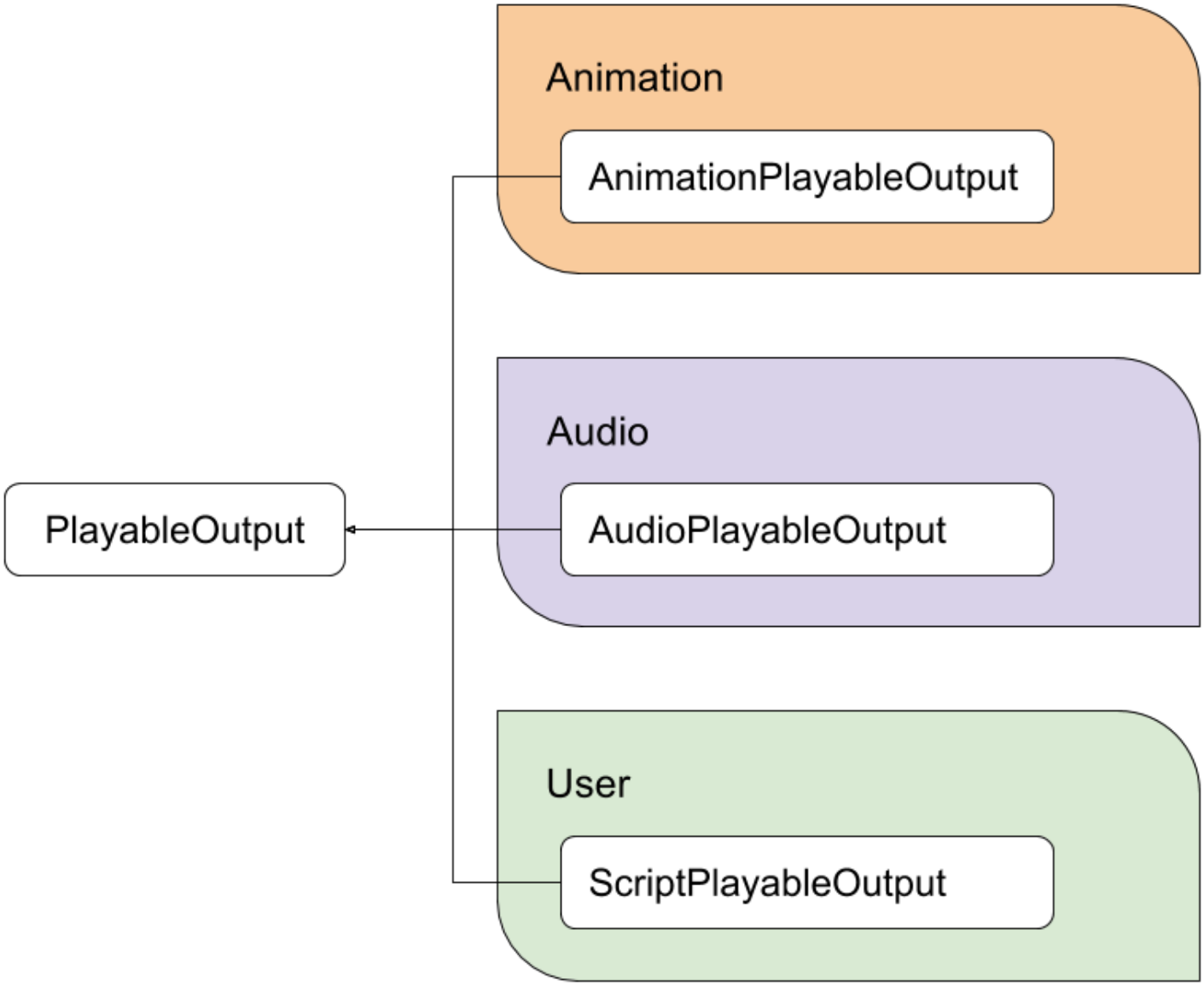


Figure 3: Core playable output types

The playable core types and playable output types are implemented as C# structs to avoid allocating memory for garbage collection.

'Playable' is the base type for all playables, meaning that you can always implicitly cast a playable to it. The opposite is not true, and an exception will be thrown if a 'Playable' is explicitly casted into an incompatible type. It also defines all the basic methods that can be executed on a playable. To access type-specific methods, you need to cast our playable to the appropriate type.

The same thing is true for 'PlayableOutput', it is the base type for all playable outputs and it defines the basic methods.

Note: `Playable` and `PlayableOutput` do not expose a lot of methods. Instead, the 'PlayableExtensions' and 'PlayableOutputExtensions' static classes provide extension methods.

All non-abstract playables have a public static method `Create()` that creates a playable of the corresponding type. The 'Create()' method always takes a PlayableGraph as its first parameter, and that graph owns the newly created playable. Additional parameters may be required for some type of playables. Non-abstract playable outputs also expose a `Create()` method.

A valid playable output should be linked to a playable. If a playable output is not linked to a playable, the playable output does nothing. To link a playable output to a playable, use the `PlayableOutput.SetSourcePlayable()` method. The linked playable acts as the root of the playable tree, for that specific playable output.

To connect two playables together, use the `PlayableGraph.Connect()` method. Note that some playables cannot have inputs.

Use the `PlayableGraph.Create()` static method to create a PlayableGraph.

Play a PlayableGraph with the `PlayableGraph.Play()` method.

Stop a playing PlayableGraph with the`PlayableGraph.Stop()` method.

Evaluate the state of a PlayableGraph, at a specific time, with the `PlayableGraph.Evaluate()` method.

Destroy a PlayableGraph manually with the `PlayableGraph.Destroy()` method. This method automatically destroys all playables and playable outputs that were created by the PlayableGraph. You must manually call this destroy method to destroy a PlayableGraph, otherwise Unity issues an error message.

2017–07–04 Page published with limited editorial review

New in Unity 2017.1

ScriptPlayable and PlayableBehaviour Leave feedback

To create your own custom playable, it must be inherited from the PlayableBehaviour base class. `public class MyCustomPlayableBehaviour : PlayableBehaviour { // Implementation of the custom playable behaviour // Override PlayableBehaviour methods as needed }`

To use a PlayableBehaviour as a custom playable, it also must be encapsulated within a ScriptPlayable<> object. If you don't have an instance of your custom playable, you can create a ScriptPlayable<> for your object by calling:

```
ScriptPlayable<MyCustomPlayableBehaviour>.Create(playableGraph);
```

If you already have an instance of your custom playable, you can wrap it with a ScriptPlayable<> by calling:

```
MyCustomPlayableBehaviour myPlayable = new MyCustomPlayableBehaviour();
ScriptPlayable<MyCustomPlayableBehaviour>.Create(playableGraph, myPlayable);
```

In this case, the instance is cloned before it is assigned to the ScriptPlayable<>. As it is, this code does exactly the same as the previous code; the difference is that `myPlayable` can be a public property that would be configured in the inspector, and you can then set up your behaviour for each instance of your script.

You can get the PlayableBehaviour object from the ScriptPlayable<> by using the `ScriptPlayable<T>.GetBehaviour()` method.

2017–07–04 Page published with limited editorial review

New in Unity 2017.1

Playables Examples

Leave feedback

PlayableGraph Visualizer

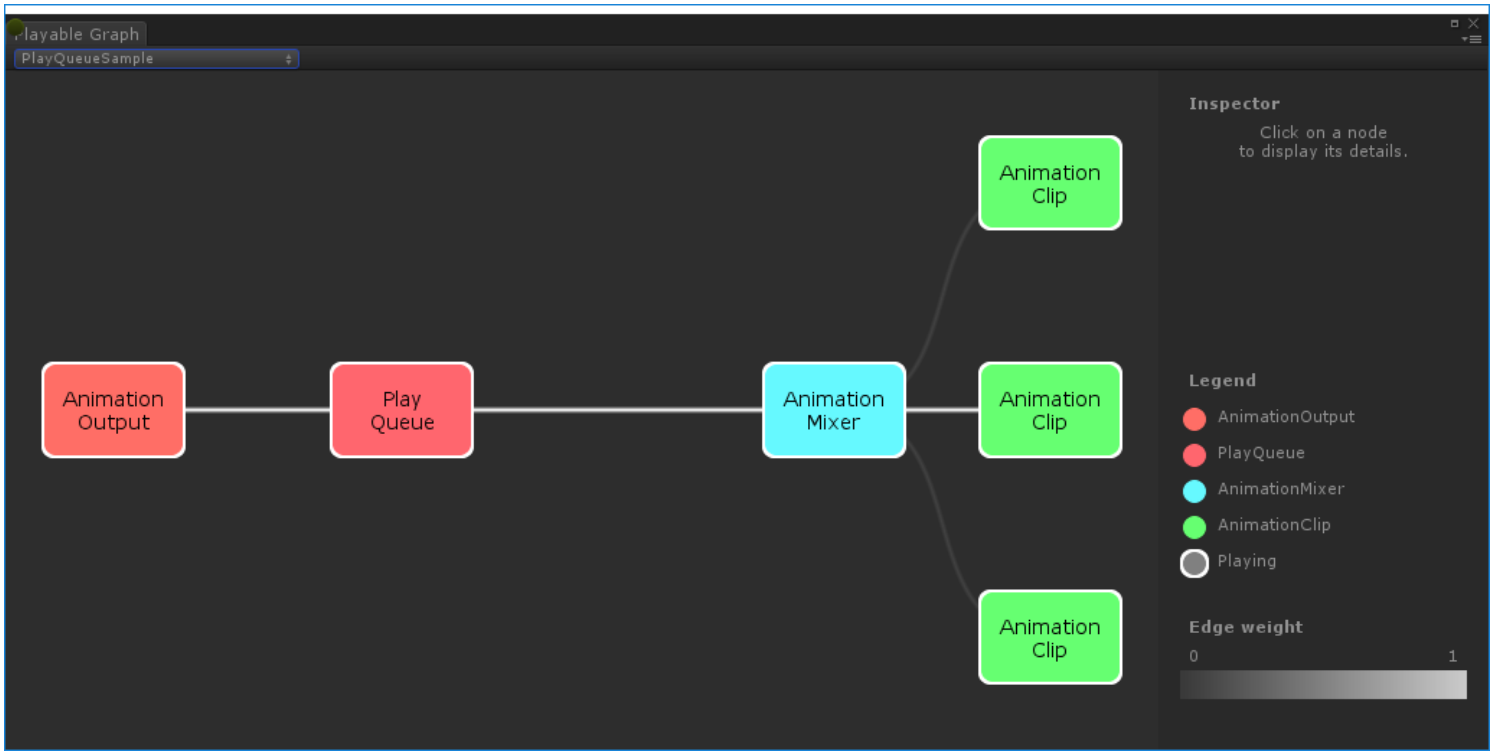
All of the examples in this document use the PlayableGraph Visualizer (Pictured below) to illustrate the trees and nodes created by the **Playables** API. The **Playable graph** Visualizer is a tool available through GitHub.

To use the PlayableGraph Visualizer:

Download the PlayableGraph Visualizer that corresponds with your version of Unity from the GitHub repository

Open the tool by selecting **Window** > **PlayableGraph Visualizer**

Register your graph using GraphVisualizerClient.Show(PlayableGraph graph, string name).



The GraphVisualizer window

Playables in the graph are represented by colored nodes. Wire color intensity indicates the weight of the blending. See GitHub for more information on this tool.

Playing a single animation clip on a GameObject

This example demonstrates a simple `PlayableGraph` with a single playable output that is linked to a single playable node. The playable node plays a single **animation clip** (clip). An `AnimationClipPlayable` must wrap the animation clip to make it compatible with the Playables API.

```
using UnityEngine;

using UnityEngine.Playables;
```

```
using UnityEngine.Animations;

[RequireComponent(typeof(Animator))]

public class PlayAnimationSample : MonoBehaviour

{

    public AnimationClip clip;

    PlayableGraph playableGraph;

    void Start()

    {

        playableGraph = PlayableGraph.Create();

        playableGraph.SetTimeUpdateMode(DirectorUpdateMode.GameTime);

        var playableOutput = AnimationPlayableOutput.Create(playableGraph, "Anim

        // Wrap the clip in a playable

        var clipPlayable = AnimationClipPlayable.Create(playableGraph, clip);

        // Connect the Playable to an output

        playableOutput.SetSourcePlayable(clipPlayable);

        // Plays the Graph.

        playableGraph.Play();

    }

    void OnDisable()

    {

        // Destroys all Playables and PlayableOutputs created by the graph.

        playableGraph.Destroy();

    }
```

```
}
```

The PlayableGraph generated by PlayAnimationSample

Use `AnimationPlayableUtilities` to simplify the creation and playback of animation playables, as shown in the following example:__ __

```
using UnityEngine;

using UnityEngine.Playables;

using UnityEngine.Animations;

[RequireComponent(typeof(Animator))]

public class PlayAnimationUtilitiesSample : MonoBehaviour

{

    public AnimationClip clip;

    PlayableGraph playableGraph;
```

```
    void Start()

    {

        AnimationPlayableUtilities.PlayClip(GetComponent<Animator>(), clip, out

    }

    void OnDisable()

    {

        // Destroys all Playables and Outputs created by the graph.

        playableGraph.Destroy();

    }

}
```

Creating an animation blend tree

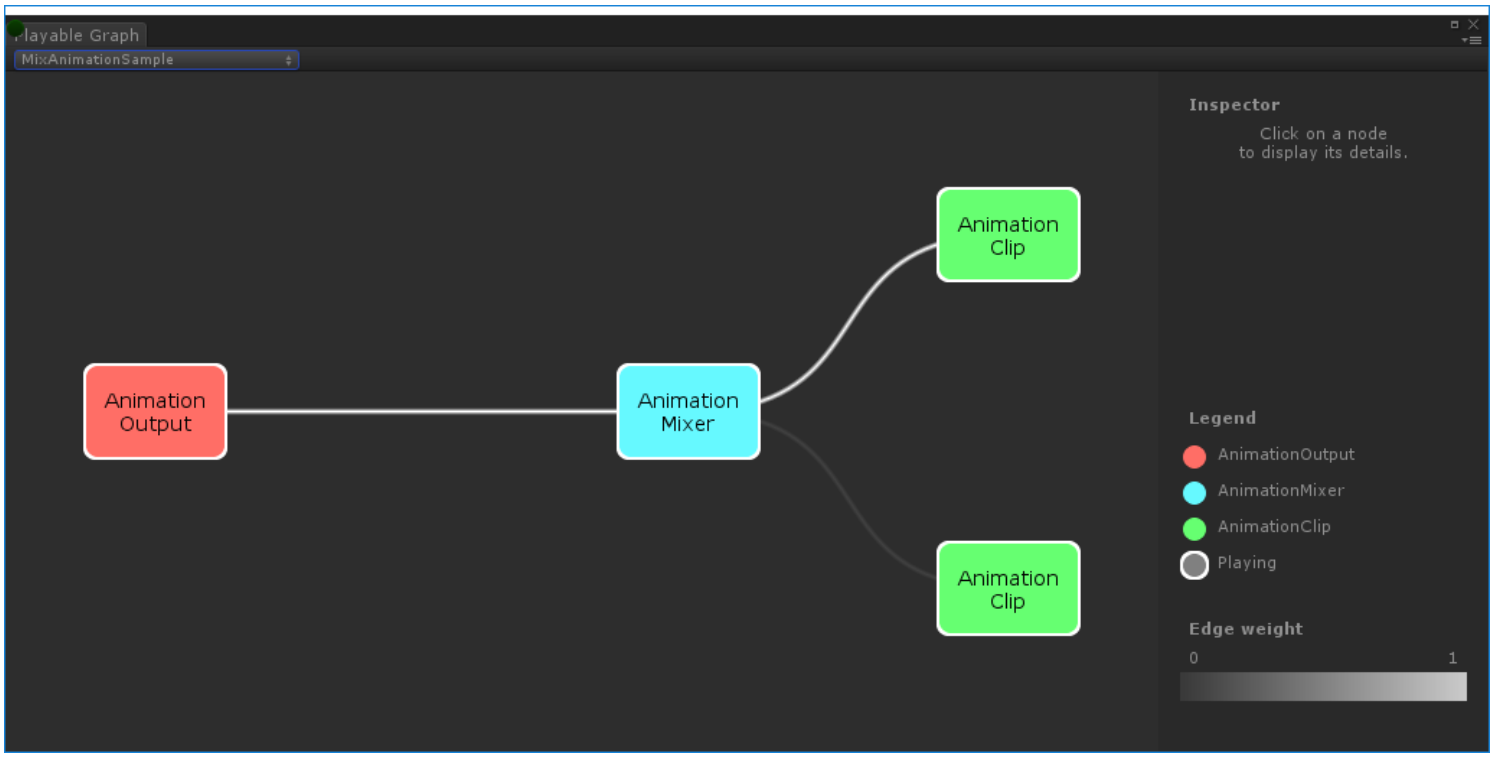
This example demonstrates how to use the `AnimationMixerPlayable` to blend two animation clips. Before blending the animation clips, they must be wrapped by playables. To do this, an `AnimationClipPlayable` (clipPlayable0 and clipPlayable1) wraps each `AnimationClip` (clip0 and clip1). The `SetInputWeight()` method dynamically adjusts the blend weight of each playable.

Although not shown in this example, you can also use `AnimationMixerPlayable` to blend playable mixers and other playables.

```
using UnityEngine;

using UnityEngine.Playables;

using UnityEngine.Animations;

[RequireComponent(typeof(Animator))]

public class MixAnimationSample : MonoBehaviour

{
```

```
public AnimationClip clip0;

public AnimationClip clip1;

public float weight;

PlayableGraph playableGraph;

AnimationMixerPlayable mixerPlayable;

void Start()

{

    // Creates the graph, the mixer and binds them to the Animator.

    playableGraph = PlayableGraph.Create();

    var playableOutput = AnimationPlayableOutput.Create(playableGraph, "Anim

    mixerPlayable = AnimationMixerPlayable.Create(playableGraph, 2);

    playableOutput.SetSourcePlayable(mixerPlayable);

    // Creates AnimationClipPlayable and connects them to the mixer.

    var clipPlayable0 = AnimationClipPlayable.Create(playableGraph, clip0);

    var clipPlayable1 = AnimationClipPlayable.Create(playableGraph, clip1);

    playableGraph.Connect(clipPlayable0, 0, mixerPlayable, 0);

    playableGraph.Connect(clipPlayable1, 0, mixerPlayable, 1);



    // Plays the Graph.

    playableGraph.Play();

}

void Update()

{
```

```
        weight = Mathf.Clamp01(weight);

        mixerPlayable.SetInputWeight(0, 1.0f-weight);

        mixerPlayable.SetInputWeight(1, weight);

    }

    void OnDisable()

    {

        // Destroys all Playables and Outputs created by the graph.

        playableGraph.Destroy();

    }

}
```



The `PlayableGraph` generated by `MixAnimationSample

Blending an AnimationClip and AnimatorController

This example demonstrates how to use an `AnimationMixerPlayable` to blend an `AnimationClip` with an `AnimatorController`.

Before blending the `AnimationClip` and `AnimatorController`, they must be wrapped by playables. To do this, an `AnimationClipPlayable` (clipPlayable) wraps the `AnimationClip` (clip) and an `AnimatorControllerPlayable` (ctrlPlayable) wraps the RuntimeAnimatorController (controller). The `SetInputWeight()` method dynamically adjusts the blend weight of each playable.

```
using UnityEngine;

using UnityEngine.Playables;

using UnityEngine.Animations;

[RequireComponent(typeof(Animator))]

public class RuntimeControllerSample : MonoBehaviour

{

    public AnimationClip clip;

    public RuntimeAnimatorController controller;

    public float weight;

    PlayableGraph playableGraph;

    AnimationMixerPlayable mixerPlayable;

    void Start()

    {

        // Creates the graph, the mixer and binds them to the Animator.

        playableGraph = PlayableGraph.Create();

        var playableOutput = AnimationPlayableOutput.Create(playableGraph, "Anim

        mixerPlayable = AnimationMixerPlayable.Create(playableGraph, 2);

        playableOutput.SetSourcePlayable(mixerPlayable);
```

```
        // Creates AnimationClipPlayable and connects them to the mixer.
        var clipPlayable = AnimationClipPlayable.Create(playableGraph, clip);
        var ctrlPlayable = AnimatorControllerPlayable.Create(playableGraph, cont
        playableGraph.Connect(clipPlayable, 0, mixerPlayable, 0);
        playableGraph.Connect(ctrlPlayable, 0, mixerPlayable, 1);

        // Plays the Graph.
        playableGraph.Play();
    }
    void Update()
    {
        weight = Mathf.Clamp01(weight);
        mixerPlayable.SetInputWeight(0, 1.0f-weight);
        mixerPlayable.SetInputWeight(1, weight);
    }
    void OnDisable()
    {
        // Destroys all Playables and Outputs created by the graph.
        playableGraph.Destroy();
    }
}
```

Creating a PlayableGraph with several outputs

This example demonstrates how to create a `PlayableGraph` with two different playable output types: an `AudioPlayableOutput` and an `AnimationPlayableOutput`. A `PlayableGraph` can have many playable outputs of different types.

This example also demonstrates how to play an `AudioClip` through an `AudioClipPlayable` that is connected to an `AudioPlayableOutput`.

```
using UnityEngine;

using UnityEngine.Animations;

using UnityEngine.Audio;

using UnityEngine.Playables;

[RequireComponent(typeof(Animator))]

[RequireComponent(typeof(AudioSource))]

public class MultiOutputSample : MonoBehaviour

{

    public AnimationClip animationClip;

    public AudioClip audioClip;

    PlayableGraph playableGraph;

    void Start()

    {

        playableGraph = PlayableGraph.Create();

        // Create the outputs.

        var animationOutput = AnimationPlayableOutput.Create(playableGraph, "Ani

        var audioOutput = AudioPlayableOutput.Create(playableGraph, "Audio", Get



        // Create the playables.
```

```
        var animationClipPlayable = AnimationClipPlayable.Create(playableGraph,

        var audioClipPlayable = AudioClipPlayable.Create(playableGraph, audioCli

        // Connect the playables to an output

        animationOutput.SetSourcePlayable(animationClipPlayable);

        audioOutput.SetSourcePlayable(audioClipPlayable);

        // Plays the Graph.

        playableGraph.Play();

    }

    void OnDisable()

    {

        // Destroys all Playables and Outputs created by the graph.

        playableGraph.Destroy();

    }

}
```

The `PlayableGraph` generated by `MultiOutputSample

Controlling the play state of the tree

This example demonstrates how to use the `Playable.SetPlayState()` method to control the play state of node on the `PlayableGraph` tree. The `SetPlayState` method controls the play state of the entire tree, one of its branches, or a single node.

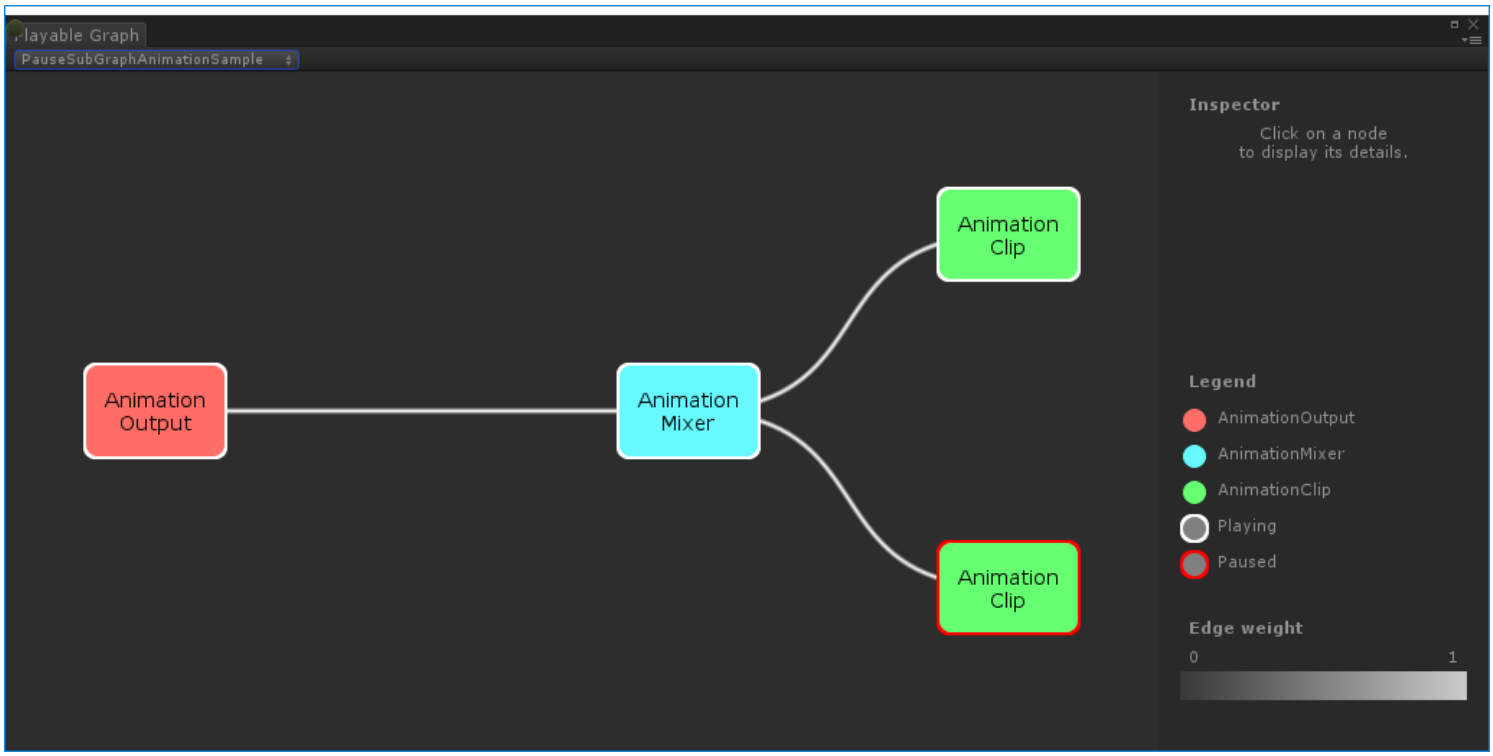
When setting the play state of a node, the state propagates to all its children, regardless of their play states. For example, if a child node is explicitly paused, setting a parent node to “playing” also sets all its child nodes to “playing.”

In this example, the `PlayableGraph` contains a mixer that blends two animation clips. An `AnimationClipPlayable` wraps each animation clip and the `SetPlayState()` method explicitly pauses the second playable. The second `AnimationClipPlayable` is explicitly paused, so its internal time does not advance and outputs the same value. The exact value depends on the specific time when the `AnimationClipPlayable` was paused.

```
using UnityEngine;

using UnityEngine.Playables;

using UnityEngine.Animations;

[RequireComponent(typeof(Animator))]

public class PauseSubGraphAnimationSample : MonoBehaviour
```

```
{

    public AnimationClip clip0;

    public AnimationClip clip1;

    PlayableGraph playableGraph;

    AnimationMixerPlayable mixerPlayable;

    void Start()

    {

        // Creates the graph, the mixer and binds them to the Animator.

        playableGraph = PlayableGraph.Create();

        var playableOutput = AnimationPlayableOutput.Create(playableGraph, "Anim

        mixerPlayable = AnimationMixerPlayable.Create(playableGraph, 2);

        playableOutput.SetSourcePlayable(mixerPlayable);

        // Creates AnimationClipPlayable and connects them to the mixer.

        var clipPlayable0 = AnimationClipPlayable.Create(playableGraph, clip0);

        var clipPlayable1 = AnimationClipPlayable.Create(playableGraph, clip1);

        playableGraph.Connect(clipPlayable0, 0, mixerPlayable, 0);

        playableGraph.Connect(clipPlayable1, 0, mixerPlayable, 1);

        mixerPlayable.SetInputWeight(0, 1.0f);

        mixerPlayable.SetInputWeight(1, 1.0f);

        clipPlayable1.SetPlayState(PlayState.Paused);

        // Plays the Graph.

        playableGraph.Play();

    }
```

```
    void OnDisable()

    {

        // Destroys all Playables and Outputs created by the graph.

        playableGraph.Destroy();

    }

}
```



The PlayableGraph generated by PauseSubGraphAnimationSample. Notice that the second clip is paused (red edge).

Controlling the timing of the tree

This example demonstrates how to use the Play() method to play a PlayableGraph, how to use the SetPlayState() method to pause a playable, and how to use the SetTime() method to manually set the local time of a playable with a variable.

```
using UnityEngine;

using UnityEngine.Playables;
```

```
using UnityEngine.Animations;

[RequireComponent(typeof(Animator))]

public class PlayWithTimeControlSample : MonoBehaviour

{

    public AnimationClip clip;

    public float time;

    PlayableGraph playableGraph;

    AnimationClipPlayable playableClip;

    void Start()

    {

        playableGraph = PlayableGraph.Create();

        var playableOutput = AnimationPlayableOutput.Create(playableGraph, "Anim

        // Wrap the clip in a playable

        playableClip = AnimationClipPlayable.Create(playableGraph, clip);

        // Connect the Playable to an output

        playableOutput.SetSourcePlayable(playableClip);

        // Plays the Graph.

        playableGraph.Play();

        // Stops time from progressing automatically.

        playableClip.SetPlayState(PlayState.Paused);

    }

    void Update ()

    {
```

```
        // Control the time manually
        playableClip.SetTime(time);
    }

    void OnDisable()
    {
        // Destroys all Playables and Outputs created by the graph.
        playableGraph.Destroy();
    }
}
```

Creating PlayableBehaviour

This example demonstrates how to create custom playables with the `PlayableBehaviour` public class. This example also demonstrate how to override the `PrepareFrame()` virtual method to control nodes on the `PlayableGraph`. Custom playables can override any of the other virtual methods of the `PlayableBehaviour` class.

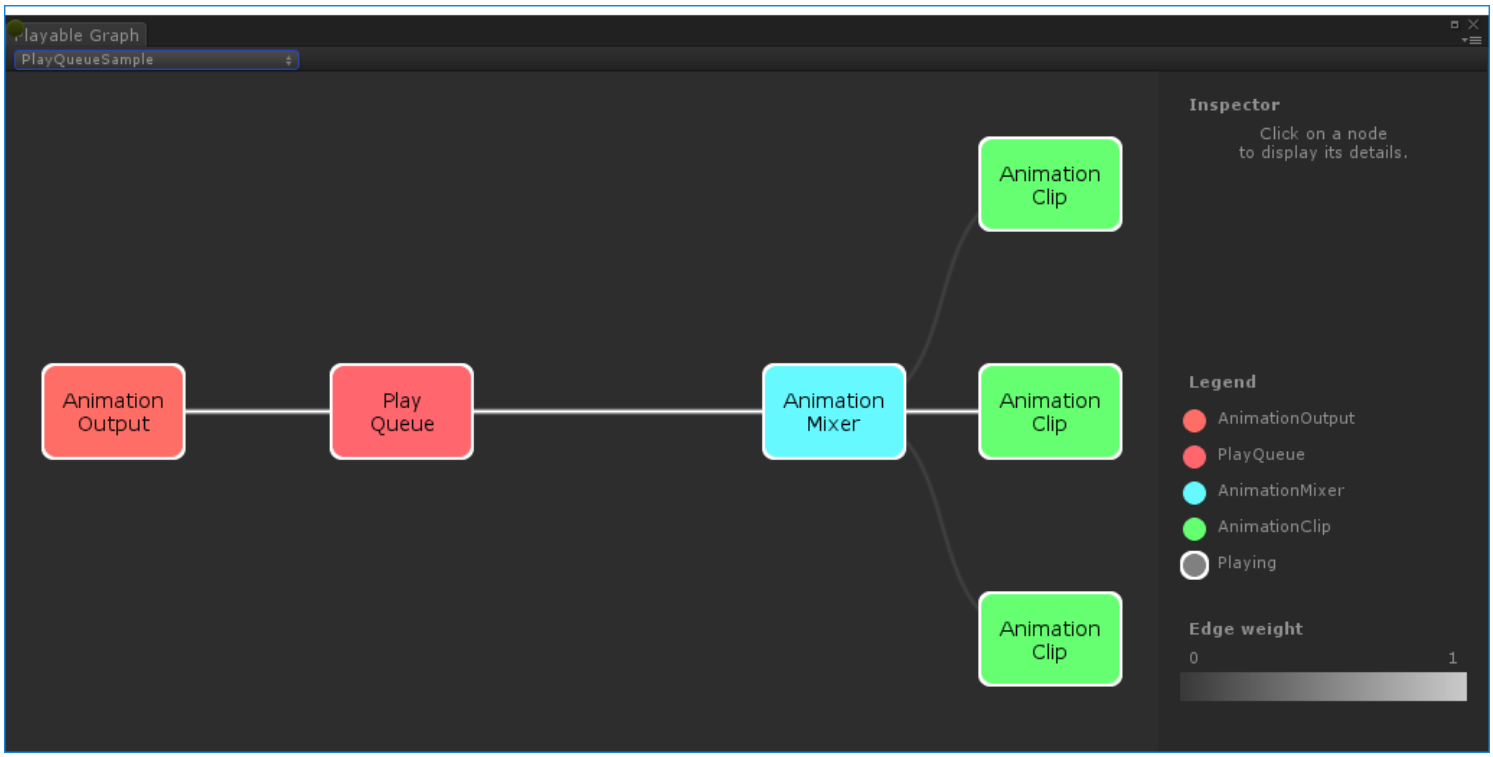
In this example, the nodes being controlled are a series of animation clips (clipsToPlay). The `SetInputMethod()` modifies the blend weight of each animation clip, ensuring that only one clip plays at a time, and the `SetTime()` method adjusts the local time so playback starts at the moment the animation clip is activated.

```
using UnityEngine;
using UnityEngine.Animations;
using UnityEngine.Playables;
public class PlayQueuePlayable : PlayableBehaviour
{
```

```
private int m_CurrentClipIndex = -1;

private float m_TimeToNextClip;

private Playable mixer;

public void Initialize(AnimationClip[] clipsToPlay, Playable owner, Playable

{

    owner.SetInputCount(1);

    mixer = AnimationMixerPlayable.Create(graph, clipsToPlay.Length);

    graph.Connect(mixer, 0, owner, 0);

    owner.SetInputWeight(0, 1);

    for (int clipIndex = 0 ; clipIndex < mixer.GetInputCount() ; ++clipIndex

    {

        graph.Connect(AnimationClipPlayable.Create(graph, clipsToPlay[clipIn

        mixer.SetInputWeight(clipIndex, 1.0f);

    }

}

override public void PrepareFrame(Playable owner, FrameData info)

{

    if (mixer.GetInputCount() == 0)

        return;

    // Advance to next clip if necessary

    m_TimeToNextClip -= (float)info.deltaTime;

    if (m_TimeToNextClip <= 0.0f)

    {
```

```
            m_CurrentClipIndex++;

            if (m_CurrentClipIndex >= mixer.GetInputCount())

                m_CurrentClipIndex = 0;

            var currentClip = (AnimationClipPlayable)mixer.GetInput(m_CurrentCli

            // Reset the time so that the next clip starts at the correct positi

            currentClip.SetTime(0);

            m_TimeToNextClip = currentClip.GetAnimationClip().length;

        }

        // Adjust the weight of the inputs

        for (int clipIndex = 0 ; clipIndex < mixer.GetInputCount(); ++clipIndex)

        {

            if (clipIndex == m_CurrentClipIndex)

                mixer.SetInputWeight(clipIndex, 1.0f);

            else

                mixer.SetInputWeight(clipIndex, 0.0f);

        }

    }

}

[RequireComponent(typeof (Animator))]

public class PlayQueueSample : MonoBehaviour

{

    public AnimationClip[] clipsToPlay;

    PlayableGraph playableGraph;
```

```
    void Start()

    {

        playableGraph = PlayableGraph.Create();

        var playQueuePlayable = ScriptPlayable<PlayQueuePlayable>.Create(playabl

        var playQueue = playQueuePlayable.GetBehaviour();

        playQueue.Initialize(clipsToPlay, playQueuePlayable, playableGraph);

        var playableOutput = AnimationPlayableOutput.Create(playableGraph, "Anim

        playableOutput.SetSourcePlayable(playQueuePlayable);
        playableOutput.SetSourceInputPort(0);

        playableGraph.Play();

    }

    void OnDisable()

    {

        // Destroys all Playables and Outputs created by the graph.

        playableGraph.Destroy();

    }

}
```



The PlayableGraph generated by PlayQueueSample

2017–07–04 Page published with limited editorial review

New in Unity 2017.1

A Glossary of animation terms

Leave feedback

Animation Clip terms

Term:	*Definition:*
Animation Clip	Animation data that can be used for animated characters or simple animations. It is a simple "unit" piece of motion, such as (one specific instance of) "Idle", "Walk" or "Run".
Animation Curves	Curves can be attached to animation clips and controlled by various parameters from the game.
Avatar Mask	A specification for which body parts to include or exclude for a skeleton. Used in Animation Layers and in the importer.

Avatar terms

Term:	*Definition:*
Avatar	An interface for retargeting one skeleton to another.
Retargeting	Applying animations created for one model to another.
Rigging	The process of building a skeleton hierarchy of bone **joints** for your **mesh**. Performed with an external tool, such as Autodesk® 3ds Max® or Autodesk® Maya®.
Skinning	The process of binding bone joints to the character's mesh or 'skin'. Performed with an external tool, such as Autodesk® 3ds Max® or Autodesk® Maya®.
Muscle definition	This allows you to have more intuitive control over the character's skeleton. When an Avatar is in place, the Animation system works in muscle space, which is more intuitive than bone space.
T-pose	The pose in which the character has their arms straight out to the sides, forming a "T". The required pose for the character to be in, in order to make an Avatar.
Bind-pose	The pose at which the character was modelled.
Human template	A pre-defined bone-mapping. Used for matching bones from FBX files to the Avatar.
Translate DoF	The three degrees-of-freedom associated with translation (movement in X,Y & Z) as opposed to rotation.

Animator and Animator Controller terms

Term:	*Definition:*
Animator Component	**Component** on a model that animates that model using the Animation system. The **component** has a reference to an Animator Controller **asset** that controls the animation.
Root Motion	Motion of character's root, whether it's controlled by the animation itself or externally.
Animator Controller	The Animator Controller controls animation through Animation Layers with Animation **State Machines** and Animation Blend Trees, controlled by Animation Parameters. The same Animator Controller can be referenced by multiple models with Animator components.
Animator Window	The window where the Animator Controller is visualized and edited.
Animation Layer	An Animation Layer contains an Animation State Machine that controls animations of a model or part of it. An example of this is if you have a full-body layer for walking or jumping and a higher layer for upper-body motions such as throwing an object or shooting. The higher layers take precedence for the body parts they control.

Term:	*Definition:*
Animation State Machine	A graph controlling the interaction of Animation States. Each state references an Animation Blend Tree or a single Animation Clip.
Animation Blend Tree	Used for continuous blending between similar Animation Clips based on float Animation Parameters.
Animation Parameters	Used to communicate between scripting and the Animator Controller. Some parameters can be set in scripting and used by the controller, while other parameters are based on Custom Curves in Animation Clips and can be sampled using the scripting API.
Inverse Kinematics (IK)	The ability to control the character's body parts based on various objects in the world.

Timeline

Leave feedback

Unity's Timeline

Use Unity's **Timeline** to create cinematic content, game-play sequences, audio sequences, and complex particle effects.

Each cut-scene, cinematic, or game-play sequence that you create with Unity's Timeline consists of a **Timeline Asset** and a **Timeline instance**. The Timeline Editor window creates and modifies Timeline **Assets** and Timeline instances simultaneously.

The Timeline Overview section includes details on the relationship between the Timeline Editor window, Timeline Assets, and Timeline instances.

The Timeline Workflows section shows how to create Timeline Assets and Timeline instances, how to record basic animation, and how to create cinematics.

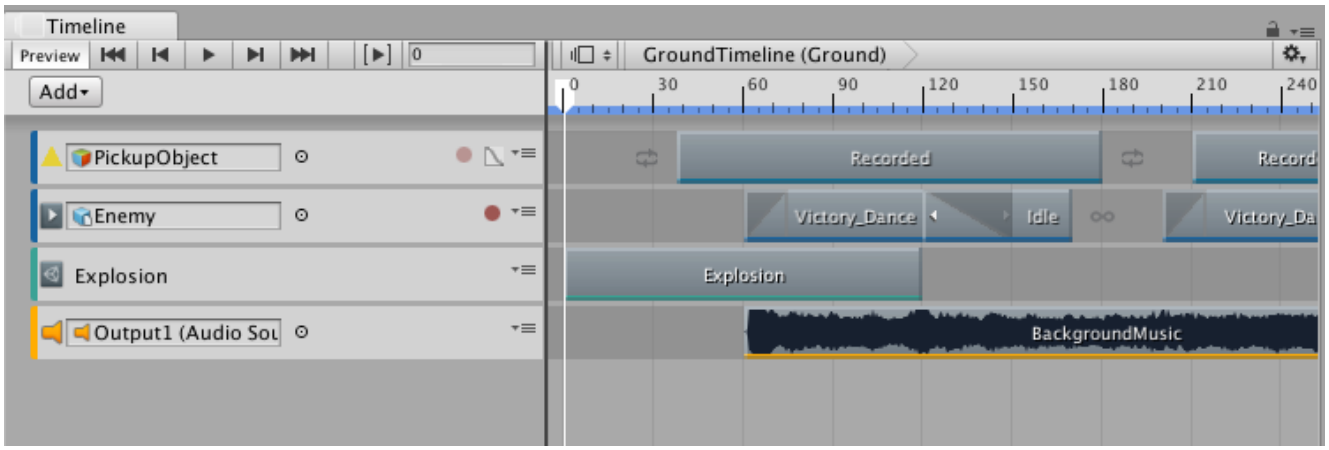
2017–08–10 Page published with limited editorial review

New feature in Unity 2017.1

Timeline overview

Leave feedback

Use the **Timeline Editor window** to create cut-scenes, cinematics, and game-play sequences by visually arranging tracks and clips linked to **GameObjects** in your **scene**.



A cinematic in the Timeline Editor window. The tracks and clips are saved to the project. The references to GameObjects are saved to the scene.

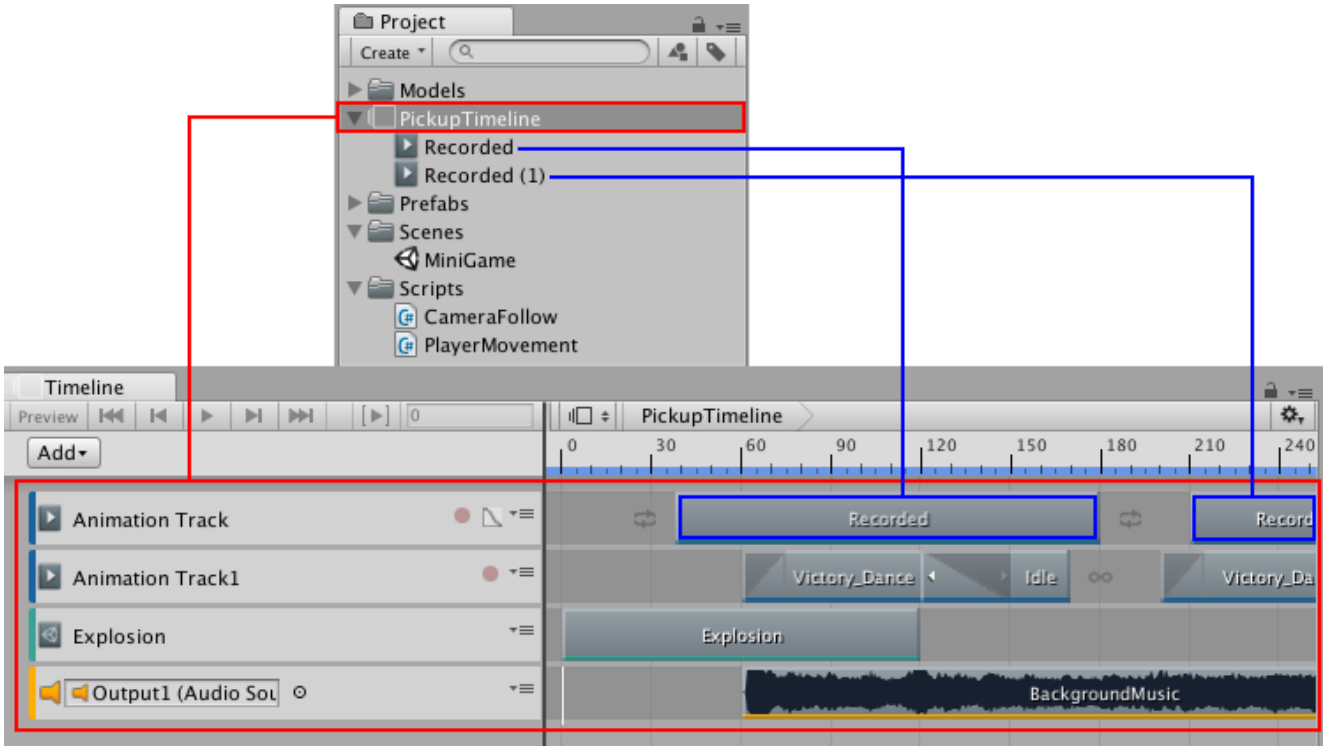
For each cut-scene, cinematic, or game-play sequence, the **Timeline** Editor window saves the following:

Timeline Asset: stores the tracks, clips, and recorded animations without links to the specific GameObjects being animated. The Timeline **Asset** is saved to the project.

Timeline instance: stores links to the specific GameObjects being animated by the Timeline Asset. These links, referred to as **bindings**, are saved to the scene.

Timeline Asset

The Timeline Editor window saves track and clip definitions as a **Timeline Asset**. If you record key animations while creating your cinematic, cut-scene, or game-play sequence, the Timeline Editor window saves the recorded animation as children of the Timeline Asset.



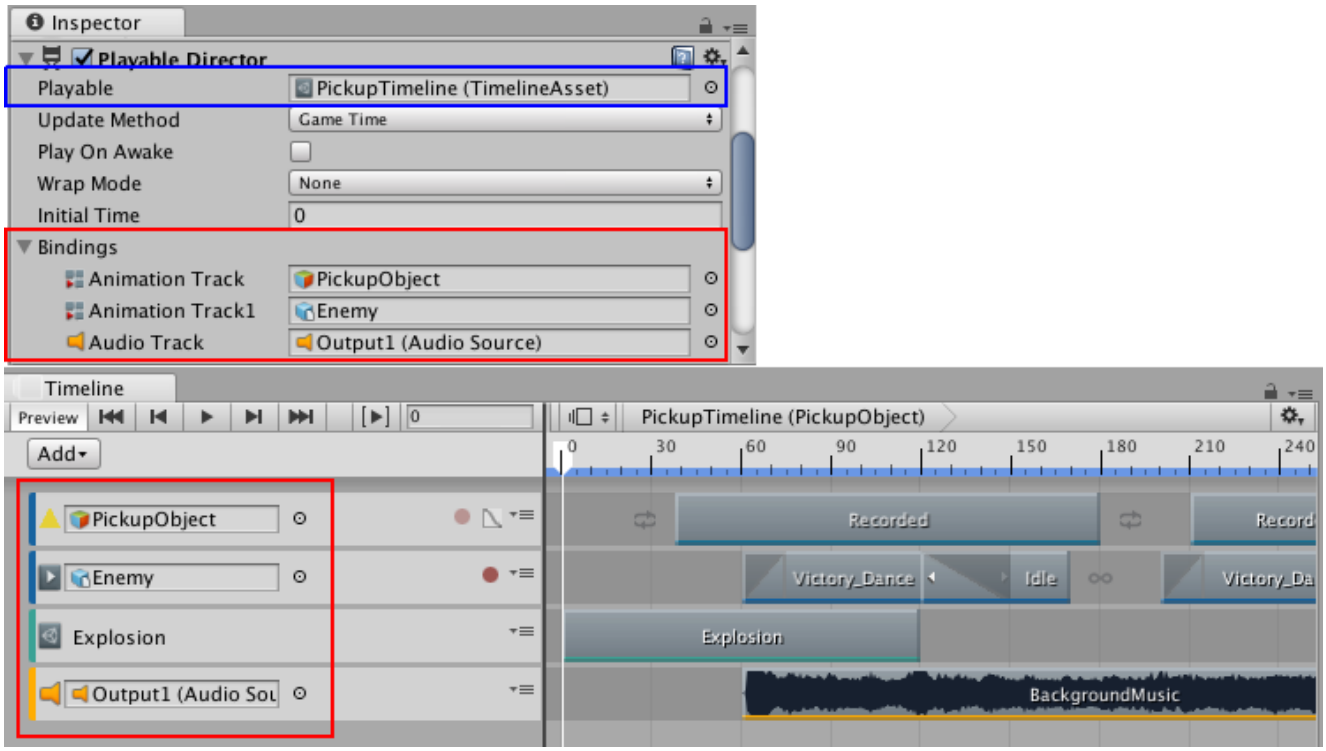
The Timeline Asset saves tracks and clips (red). If your record key animation, the recorded clips are saved as children of the Timeline Asset (blue).

Timeline instance

Although a Timeline Asset defines the tracks and clips for a cut-scene, cinematic, or game-play sequence, you cannot add a Timeline Asset directly to a scene. To animate GameObjects in your scene with a Timeline Asset, you must create a **Timeline instance**.

The Timeline Editor window provides an automated method of creating a Timeline instance while creating a Timeline Asset.

If you select a GameObject in the scene that has a Playable Director **component** associated with a Timeline Asset, the bindings appear in the Timeline Editor window and in the Playable Director **component** (Inspector window).

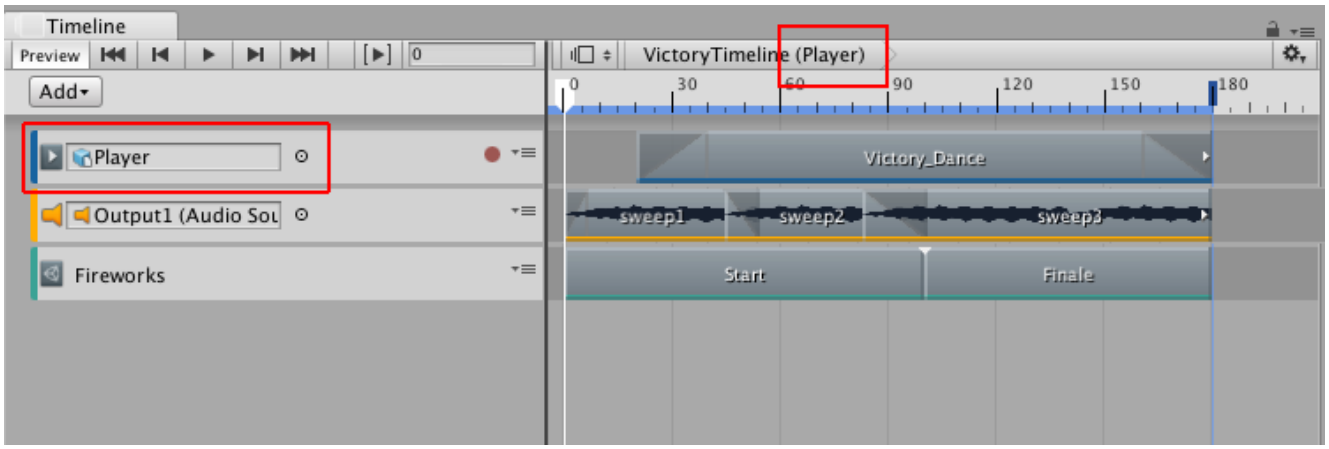


The Playable Director component shows the Timeline Asset (blue) with its bound GameObjects (red). The Timeline Editor window shows the same bindings (red) in the Track list.

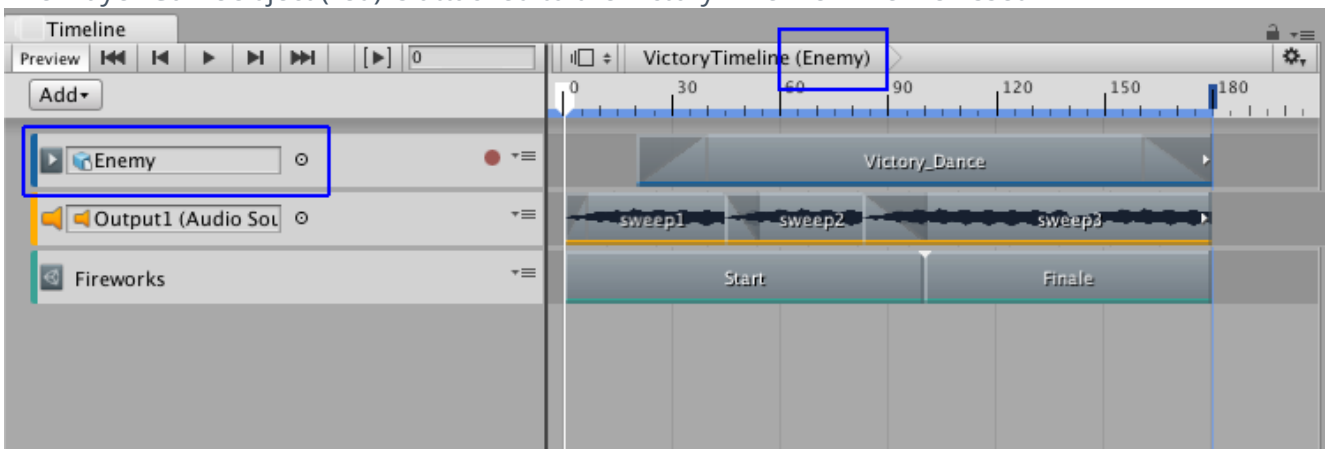
Reusing Timeline Assets

Since Timeline Assets and Timeline instances are separate, it is possible to reuse the same Timeline Asset with many Timeline instances.

For example, you can create a Timeline Asset named VictoryTimeline with the animation, music, and particle effects that play when the main game character (Player) is victorious. To reuse the VictoryTimeline Timeline Asset to animate another game character (Enemy) in the same scene, you can create another Timeline instance for the secondary game character.



The Player GameObject (red) is attached to the VictoryTimeline Timeline Asset



The Enemy GameObject (blue) is also attached to the VictoryTimeline Timeline Asset

Since the Timeline Asset is being reused, any modification to the Timeline Asset in the Timeline Editor window results in changes to all Timeline instances.

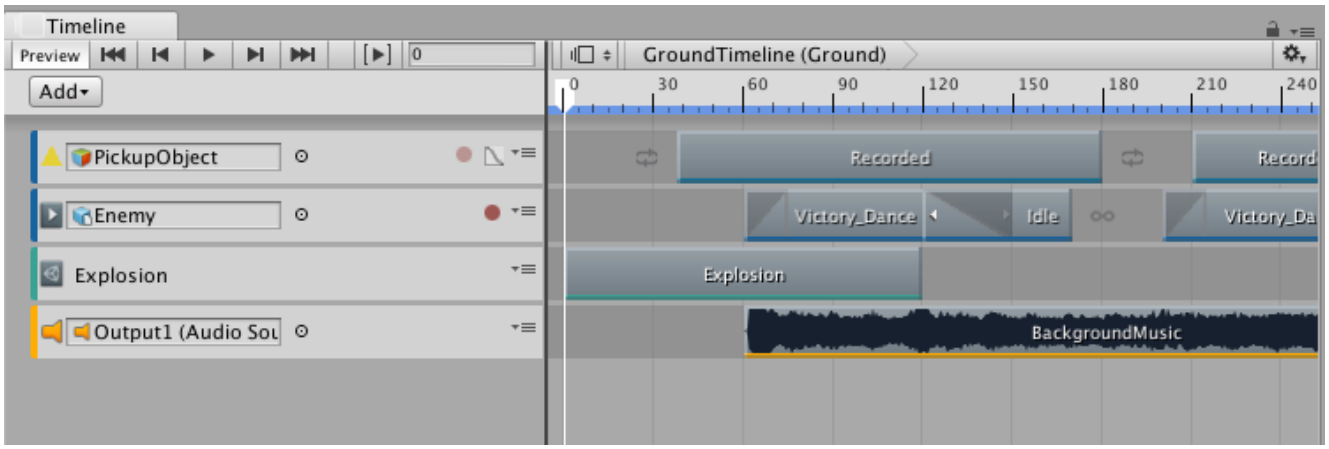
For example, in the previous example, if you delete the Fireworks Control track in the Timeline Editor window while modifying the Player Timeline instance, the track is removed from the VictoryTimeline Timeline Asset. This also removes the Fireworks control track from all instances of the VictoryTimeline Timeline Asset, including the Enemy Timeline instance.

What's the difference between the Animation window and the Timeline window?

The Timeline window

The Timeline window allows you to create cinematic content, game-play sequences, audio sequences and complex particle effects. You can animate many different GameObjects within the same sequence, such as a cut scene or scripted sequence where a character interacts with scenery. In the timeline window you can have multiple types of track, and each track can contain multiple clips that can be moved, trimmed, and blended between. It is useful for creating more complex animated sequences that require many different GameObjects to be choreographed together.

The Timeline window is newer than the Animation window. It was added to Unity in version 2017.1, and supercedes some of the functionality of the Animation window. To start learning about Timeline in Unity, visit the Timeline section of the user manual.



The Timeline window, showing many different types of clips arranged in the same sequence

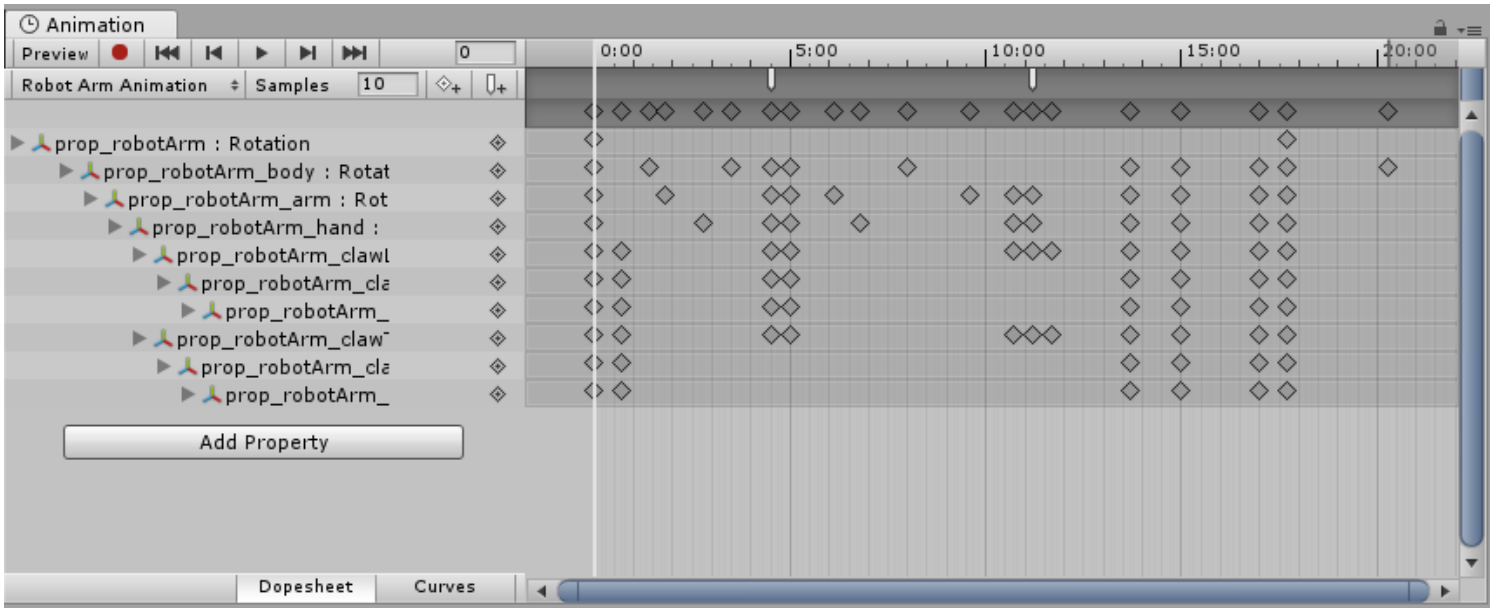
The Animation window

The Animation window allows you to create individual animation clips as well as viewing imported animation clips. Animation clips store animation for a single GameObject or a single hierarchy of GameObjects. The Animation window is useful for animating discrete items in your game such as a swinging pendulum, a sliding door, or a spinning coin. The animation window can only show one animation clip at a time.

The Animation window was added to Unity in version 4.0. The Animation window is an older feature than the Timeline window. It provides a simple way to create animation clips and animate individual GameObjects, and the clips you create in the Animation window can be combined and blended between using anim Animator controller. However, to create more complex sequences involving many disparate GameObjects you should use the Timeline window (see above).

The animation window has a “timeline” as part of its user interface (the horiontal bar with time delineations marked out), however this is separate to the Timeline window.

To start learning about animation in Unity, visit the Animation section of the user manual.



The Animation window, shown in dopesheet mode, showing a hierarchy of objects (in this case, a robot arm with numerous moving parts) animated together in a single animation clip

2017–08–10 Page published with limited editorial review

Timeline workflows

Leave feedback

The **Timeline** Editor window provides many different workflows for creating **Timeline Assets** and instances, recording animation, scheduling animation, and creating cinematic content. This section documents the following workflows:

Creating a Timeline Asset and Timeline Instance

Recording basic animation with an Infinite clip

Converting an Infinite clip to an Animation clip

Creating humanoid animation

Using Animation Override Tracks and Avatar Masking

2017–12–07 Page amended with limited editorial review

Creating a Timeline Asset and Timeline instance

Leave feedback

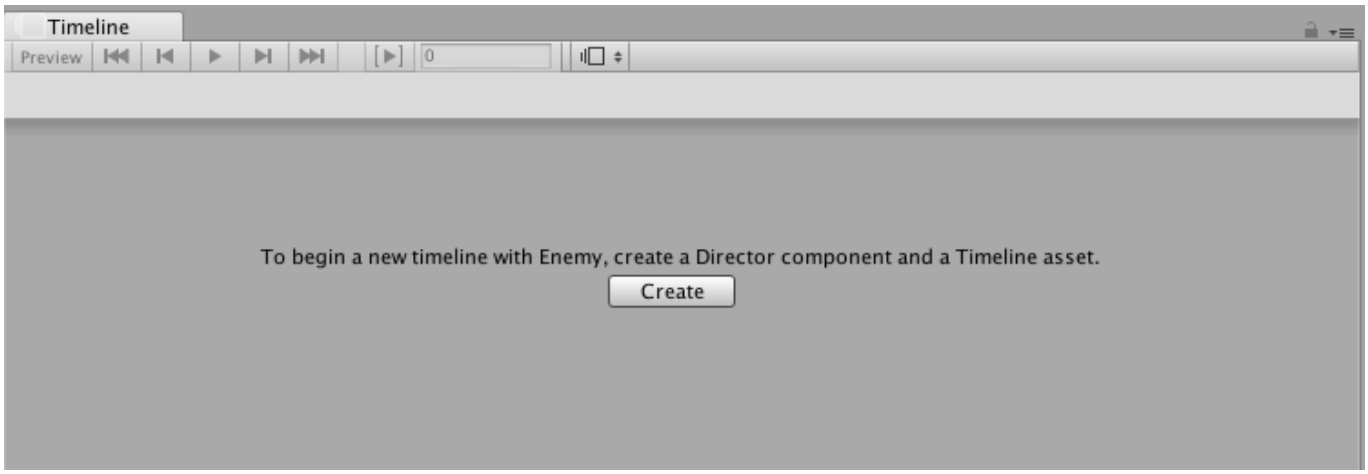
To use a **Timeline Asset** in your **scene**, associate the **Timeline Asset** to a **GameObject** using a Playable Director **component**. Associating a **Timeline Asset** with a Playable Director **component** creates a **Timeline instance** and allows you to specify which objects in the **scene** are animated by the **Timeline Asset**. The **GameObject** must also have an Animator **component**.

The **Timeline Editor window** provides an automated method of creating a Timeline instance while creating a new Timeline **Asset**. The **Timeline Editor window** also creates all the necessary components.

To create a new Timeline Asset and Timeline instance, follow these steps:

In your scene, select the GameObject that you want to use as the focus of your cinematic or other gameplay-based sequence.

Open the Timeline Editor window (menu: **Window** > **Sequencing** > **Timeline**). If the GameObject does not yet have a Playable Director component attached to a Timeline Asset, a message in the Timeline Editor window prompts you to click the **Create** button.



Timeline Editor window when selecting a GameObject not already attached to a Timeline Asset

Click **Create**. A dialog box prompts you for the name and location of the Timeline Asset being created. You can also specify tags to identify the Timeline Asset.

Click **Save**.

Unity does the following:

Saves a new Timeline Asset to the Project. If you did not change the name and location of the Timeline Asset being created, the name of the Timeline Asset is based on the selected GameObject with the “Timeline” suffix. For example, selecting the GameObject named “Enemy”, by default, names the asset “EnemyTimeline” and saves it to the Assets directory in your project.

Adds an empty Animation track to the Timeline Asset.

Adds a Playable Director component to the selected GameObject and sets the **Playable** property to the Timeline Asset. This creates a Timeline instance.

In the Playable Director component, the binding for the Animation track is set to the selected GameObject. The Animation track does not have any clips, so the selected GameObject is not animated.

Adds an **Animator component** to the selected GameObject. The **Animator component** animates the GameObject through the Timeline instance. The GameObject cannot be animated without an **Animator component**.

2017–08–10 Page published with limited editorial review

Recording basic animation with an Infinite clip

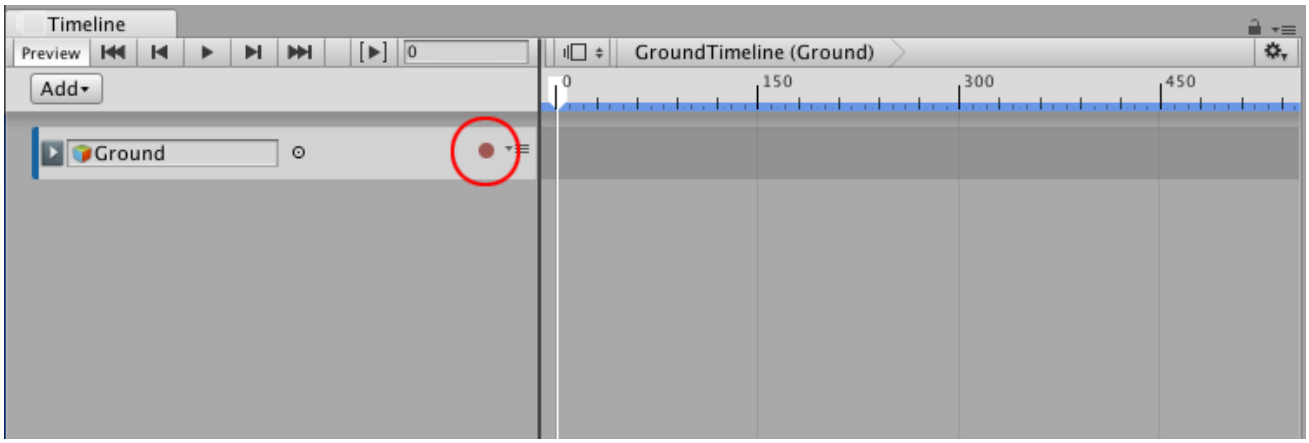
Leave feedback

You can record animation directly to an Animation track. When you record directly to an empty Animation track, you create an **Infinite clip**.

An Infinite clip is defined as a clip that contains basic key animation recorded through the **Timeline** Editor window. An Infinite clip cannot be positioned, trimmed, or split because it does not have a defined size: it spans the entirety of an Animation track.

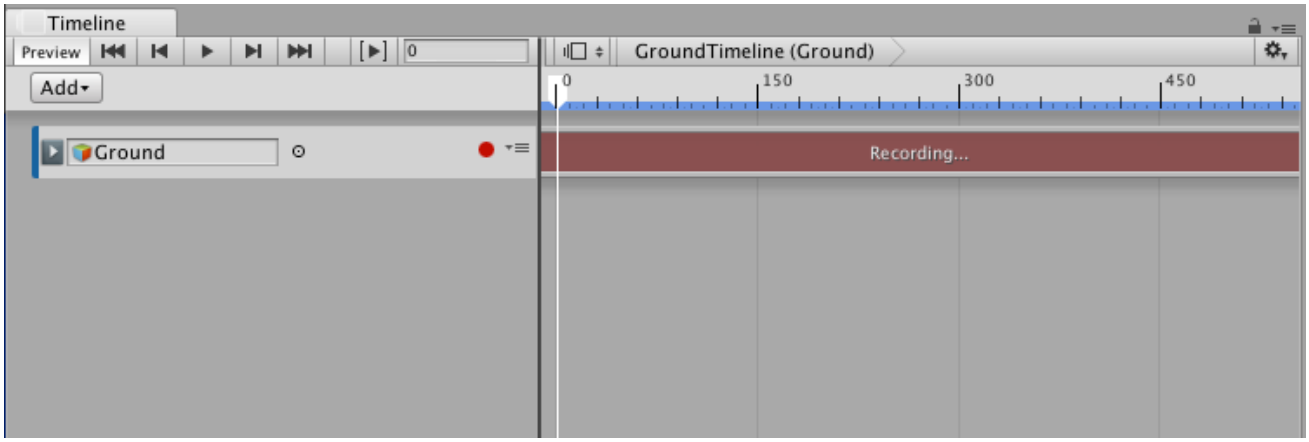
Before creating an Infinite clip, you must add an empty Animation track for the **GameObject** that you want to animate.

In the **Track list**, click the Record button for the empty Animation track to enable Record mode. The Record button is available for Animation tracks bound to simple GameObjects such as cubes, spheres, lights, and so on. The Record button is disabled for Animation tracks bound to humanoid GameObjects.



Click the Record button on an empty track to enable Record mode

When a track is in Record mode, the clip area of the track is drawn in red with the “Recording...” message. The Record button blinks.



Timeline Editor window in Record mode

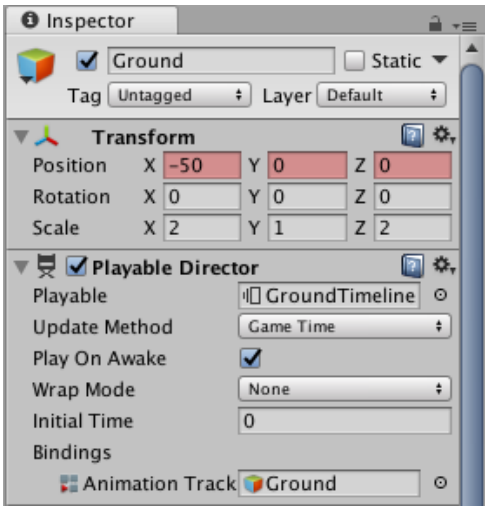
When in Record mode, any modification to an **animatable property** of the GameObject sets a key at the location of the **Timeline Playhead**. Animatable properties include transforms and the animatable properties for all **components** added to the GameObject.

To start creating an animation, move the Timeline Playhead to the location of the first key, and do one of the following:

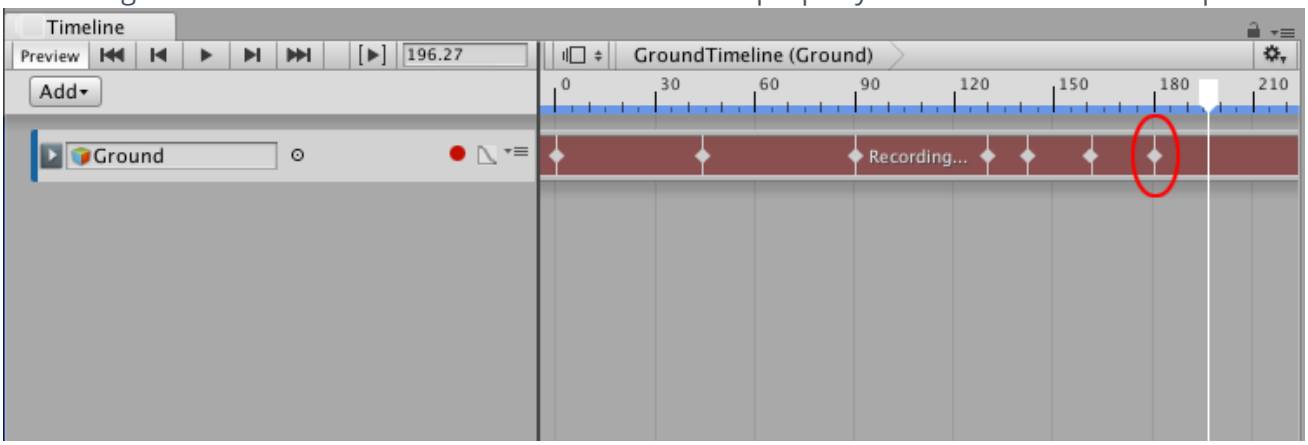
In the **Inspector** window, Right-click the name of the property and choose **Add Key**. This adds an **animation key** for the property without changing its value. A white diamond appears in the infinite clip to show the position of the key.

In the Inspector window, change the value of the animatable property of the GameObject. This adds an animation key for the property with its changed value. A white diamond appears in the infinite clip.

In the **Scene view**, move, rotate, or scale the GameObject to add a key. This automatically adds a key for the properties you change. A white diamond appears in the infinite clip.



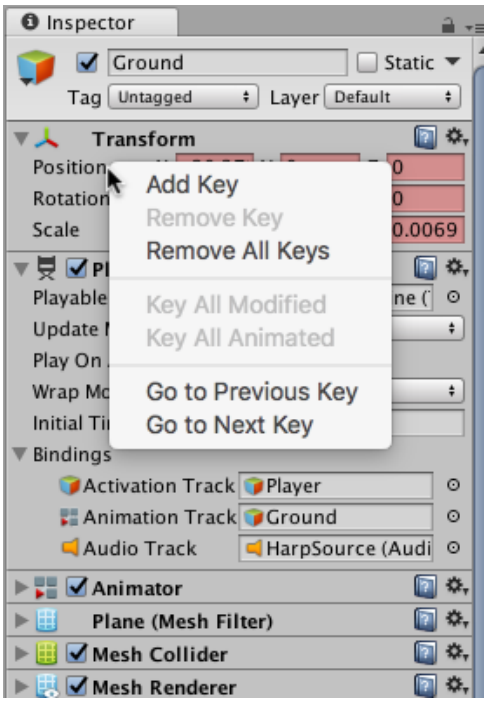
Red background indicates that an animation curve for the property has been added to the clip



Setting a key adds a white diamond to the infinite clip

Move the playhead to a different position on the timeline and change the animatable properties of the GameObject. At each position, the **Timeline Editor window** adds a white diamond to the Infinite clip for any changed properties and adds a key to its associated **animation curves**.

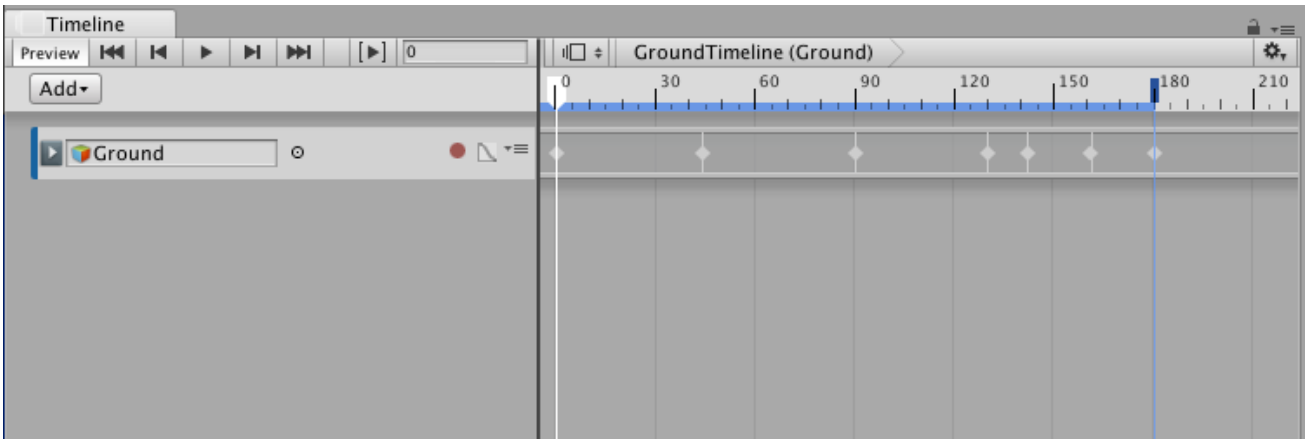
While in Record mode, you can Right-click the name of an animatable property name to perform keying operations such as setting a key without changing its value, jumping to the next or previous keys, removing keys, and so on. For example, to set a key for the position of a GameObject without changing its value, Right-click **Position** and select **Add Key** from the context menu.



Right-click the name of an animatable property to perform keying operations

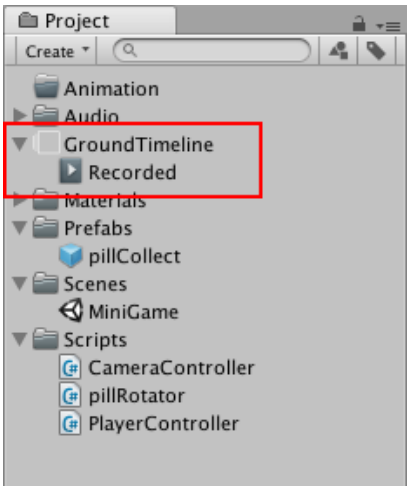
When you finish the animation, click the blinking Record button to disable Record mode.

An Infinite clip appears as a dopesheet in the Timeline Editor window, but you cannot edit the keys in this view. Use the Curves view to edit keys. You can also Double-click the Infinite clip and edit the keys with the Animation window.



An Infinite clip appear as a dopesheet

Save the **scene** or project to save the **Timeline Asset** and the Infinite clip. The Timeline Editor window saves the key animation from the Infinite clip as a source **asset**. The source **asset** is named “Recorded” and saved as a child of the **Timeline Asset** in the project.



Recorded clips are saved under the Timeline Asset in the project

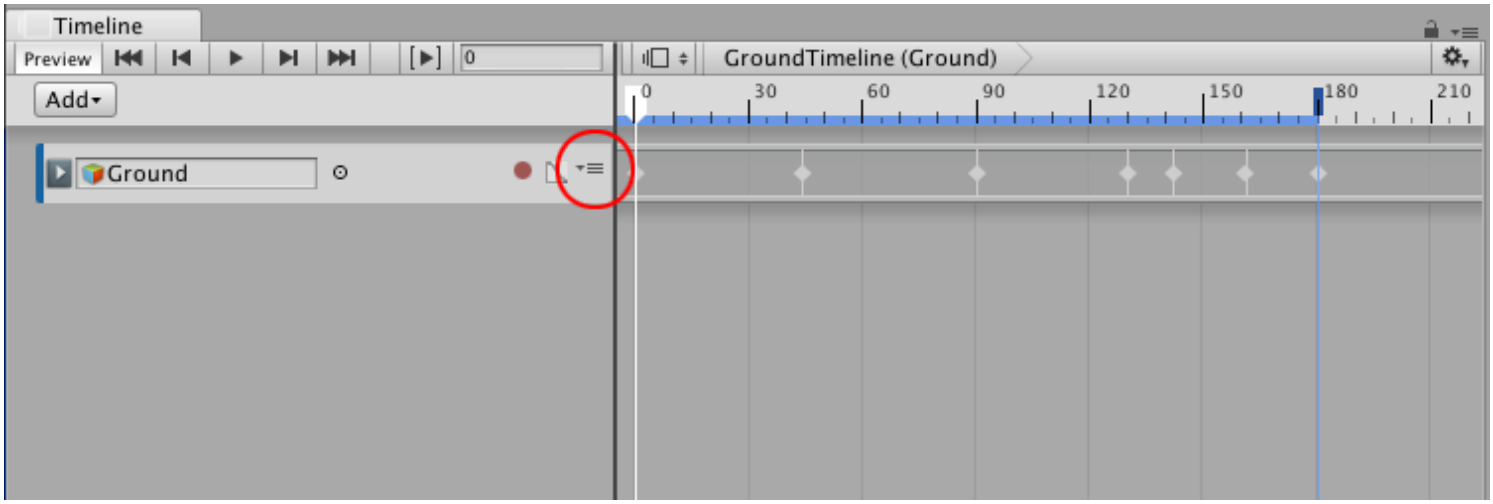
For every additional recorded Infinite clip, each clip is sequentially numbered starting at “(1)”. For example, a Timeline Asset with three recorded Infinite clips are named “Recorded”, “Recorded (1)”, and “Recorded (2)”. If you delete a Timeline Asset, its child clips are also removed.

2017–08–10 Page published with limited editorial review

Converting an Infinite clip to an Animation clip

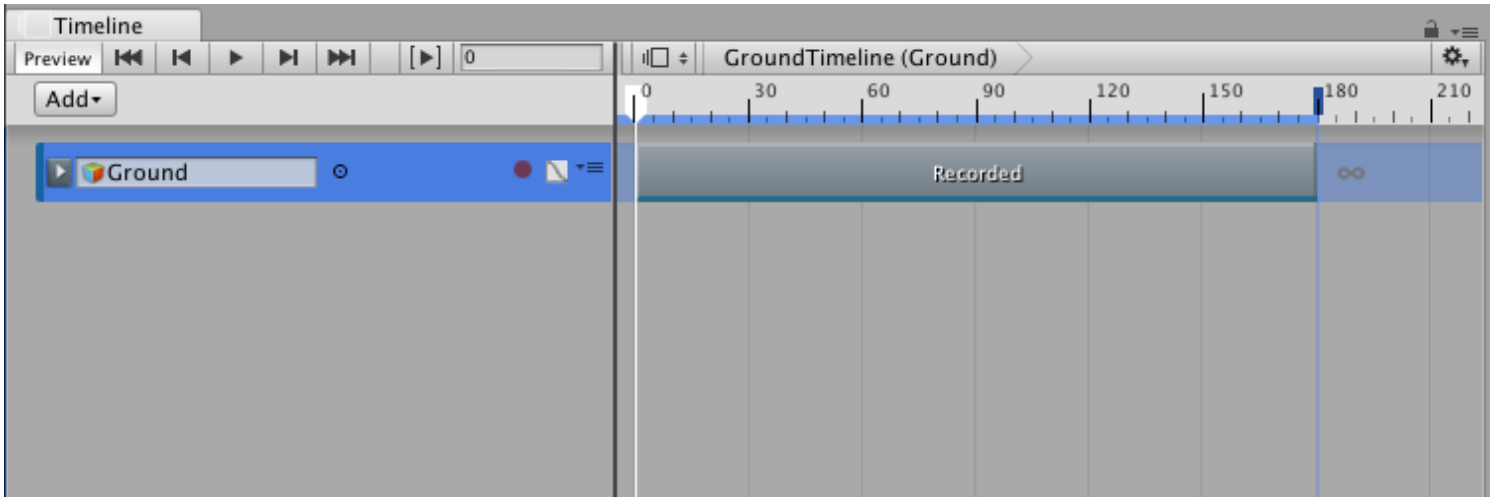
Leave feedback

An **Infinite clip** appears as a dopesheet. An **Infinite clip** cannot be positioned, trimmed, or split because it does not have a defined size. To position, trim, split, or perform other clip manipulations on an **Infinite clip**, it must first be converted to an **Animation clip**.



An infinite clip cannot be positioned, trimmed, or split. Click the Track menu (circled) to convert an Infinite clip.

To convert an Infinite clip to an Animation clip, click the Track menu icon and select **Convert to Clip Track**. You can also Right-click the track and select **Convert to Clip Track** from the context menu. The Track menu and context menu are the same.



An infinite clip after it has been converted to an Animation clip

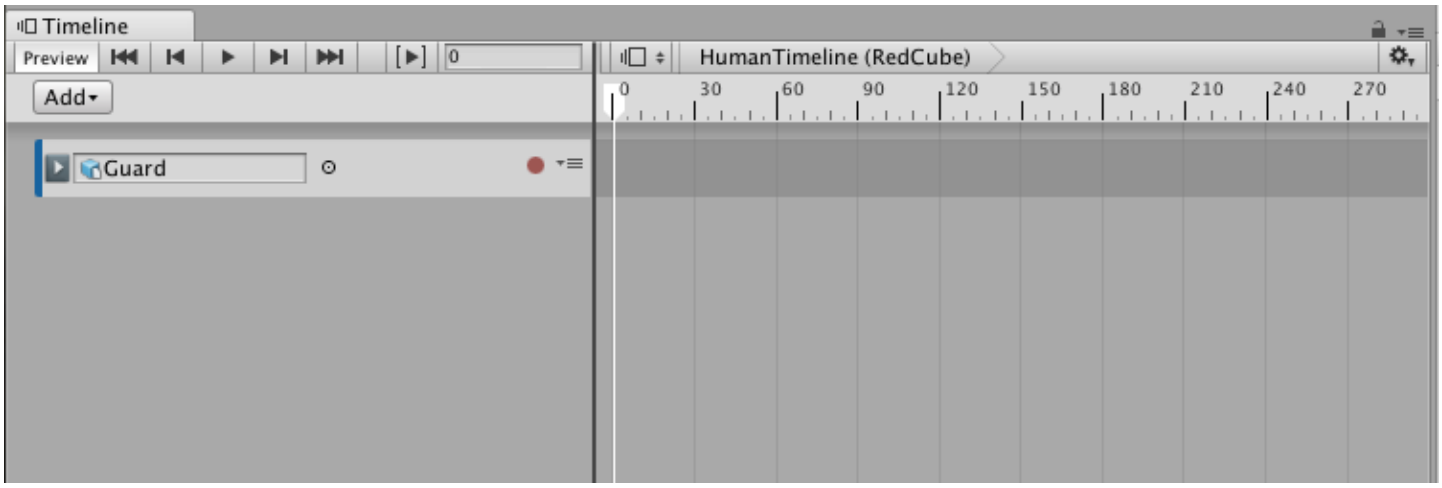
2017–08–10 Page published with limited editorial review

Creating humanoid animation

Leave feedback

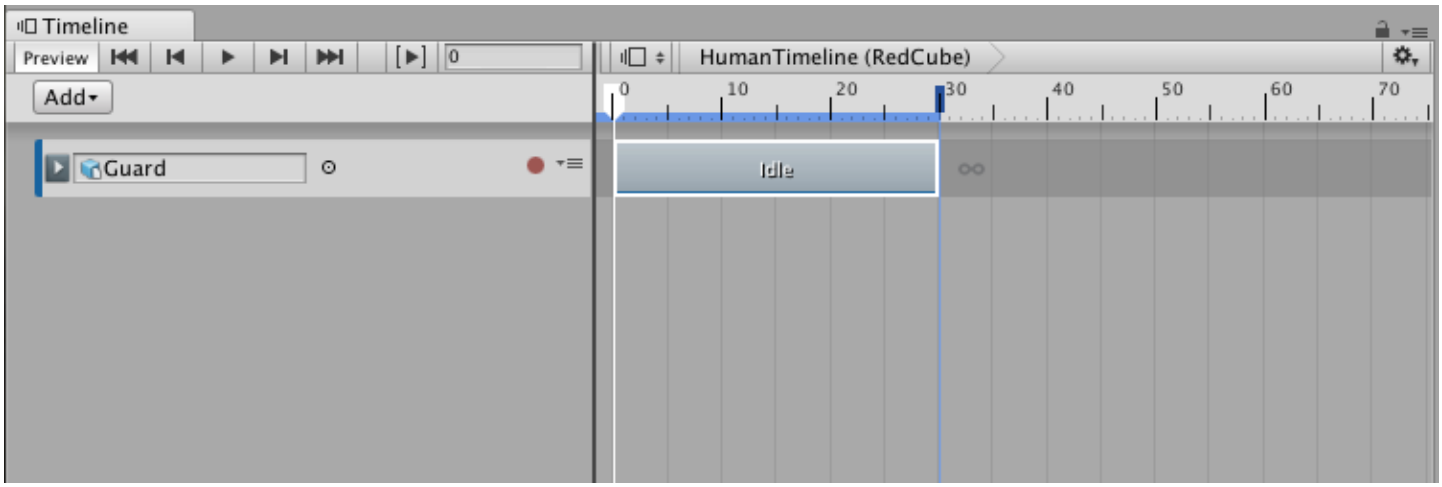
This workflow demonstrates how to use a **Timeline Instance** to animate a humanoid character with external motion clips. This workflow also demonstrates how to match clip offsets, manually adjust clip offsets, and create blends between clips to minimize jumping and sliding. Although this workflow uses a humanoid character, you can use this animation method for any **GameObject**.

This workflow assumes that you have already created a **Timeline** instance with an empty Animation track bound to a humanoid.



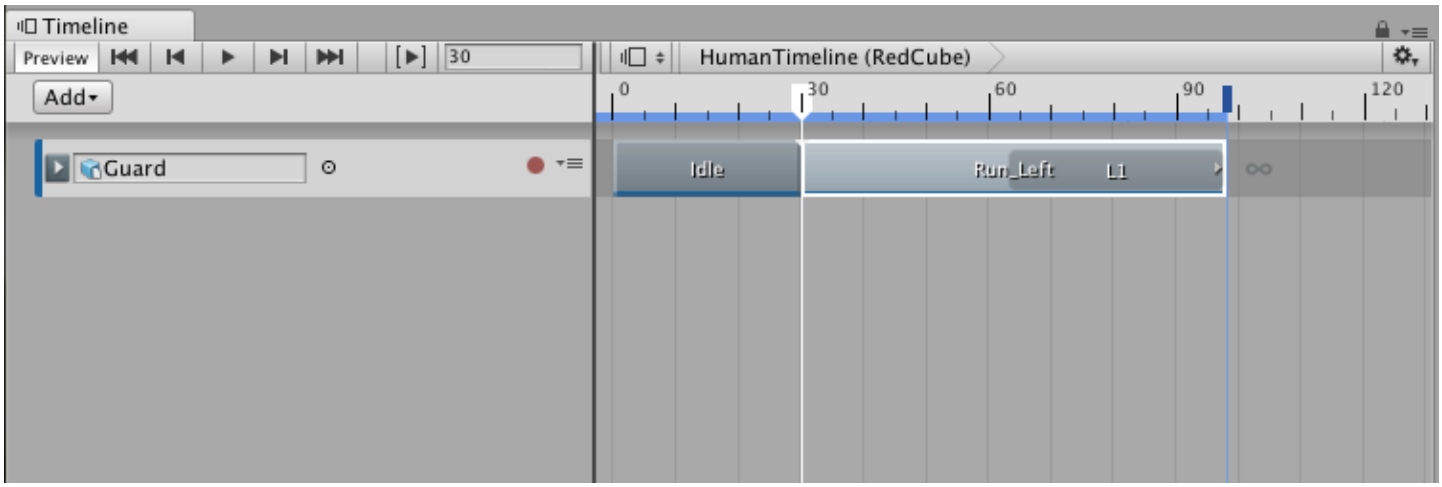
For example, the Guard humanoid is bound to an empty Animation track

From your project, drag a motion clip into the Animation track to create a new **Animation clip**. For example, drag an idle pose as the first clip to start the humanoid from an idle position. Position and resize the idle clip as appropriate.



Animation track, bound to the Guard humanoid, with an idle pose (Idle) as its Animation clip

Add a second motion clip. In this example, a run and turn left clip (named Run_Left) is dragged onto the Animation track. Resize the Run_Left clip as appropriate. In this example, the Run_Left clip is resized to include one loop so that the Guard runs and turns 180 degrees.

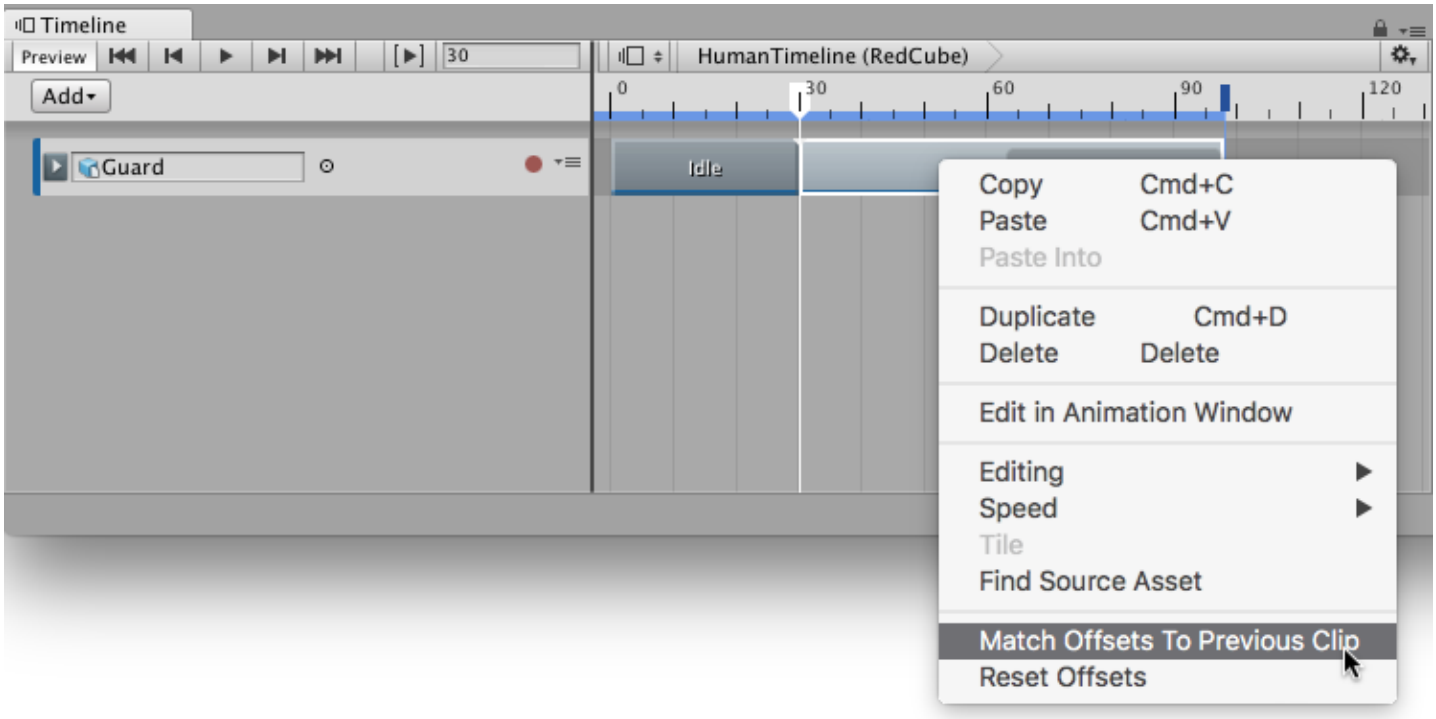


Animation track with an Idle clip and a Run_Left clip

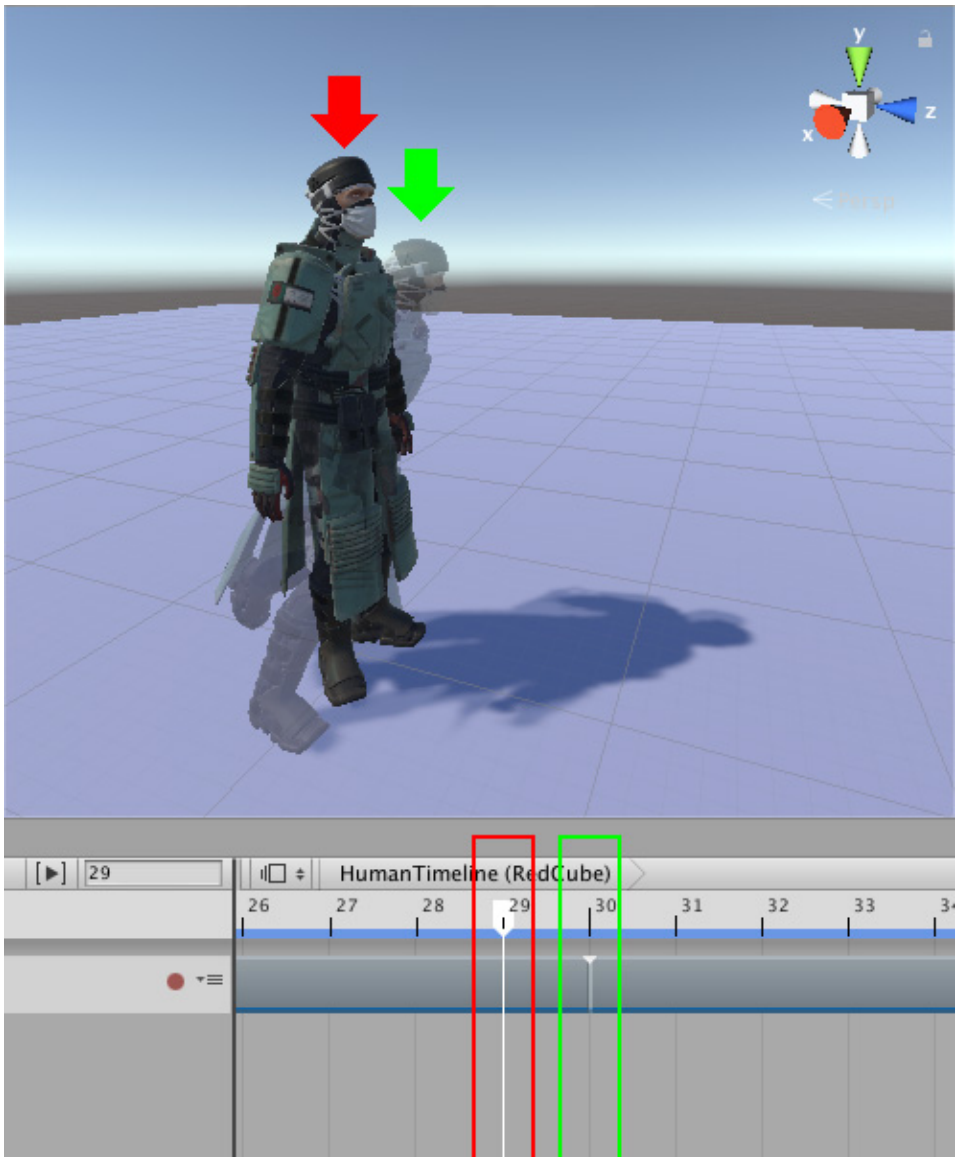
Play the Timeline instance. Notice that the humanoid, Guard character, jumps between each Animation clip. This happens because the position of the humanoid character at the end of the first Animation clip (Idle) does not match the position at the start of the next Animation clip (Run_Left).

The humanoid jumps between the first Animation clip, that ends at frame 29 (red arrow and box), and the second Animation clip, that starts at frame 30 (ghost with green arrow and box)

To fix the jump between clips, match the offset of each Animation clip. The **Timeline Editor window** provides a few different methods for matching offsets. In this example, the second Animation clip is matched with the previous clip. To do this, select the Run_Left clip, Right-click and select Match Offsets to Previous Clip.



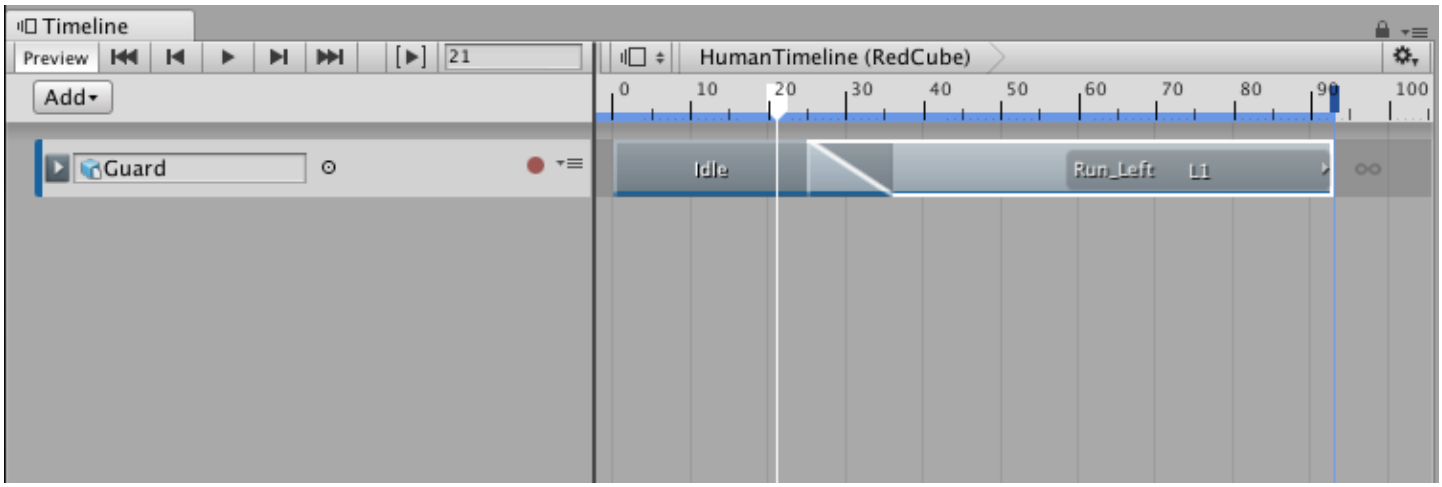
Right-click and select Match Offsets to Previous Clip to match the offsets of the selected Animation clip with the preceding Animation clip



After matching offsets, the humanoid at the end of the first Animation clip (frame 29, red arrow) matches the position and rotation of the humanoid at the start of the second Animation clip (frame 30, ghost with green arrow)

Play the Timeline instance again. Although the position and rotation of the humanoid matches, there is still a jump between the two Animation clips because the humanoid is in different poses. At the end of the first Animation clip, the humanoid is standing upright with its feet together. At the start of the second Animation clip, the humanoid is bent forward with its feet apart.

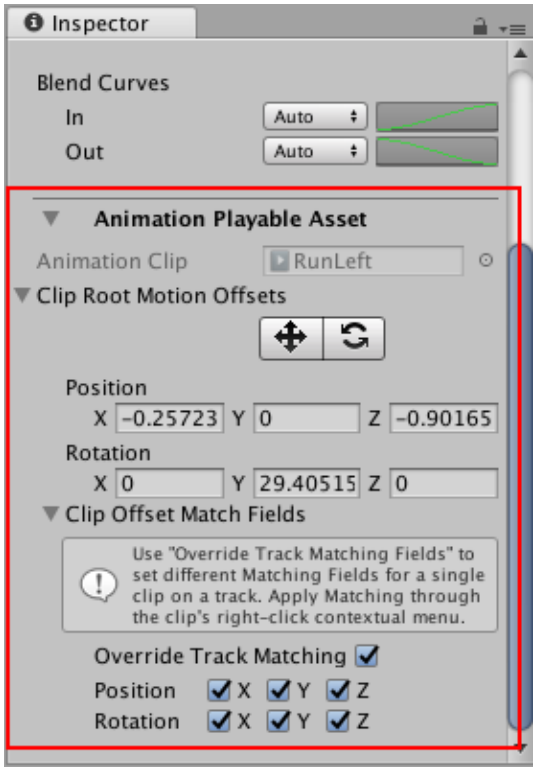
Create a blend to remove the jump and transition between the two poses. Adjust the size of the clips, the **Blend Area**, the Clip In, and the shape of each Blend Curve to create a transition between the two poses. For example, in the transition between the Idle clip and the Run_Left clip, the Idle clip is changed to a duration of 36 and the Run_Left clip is repositioned to start at frame 25. The rest of the properties are left as their default values.



Create a blend between the first two clips to create a smooth transition between the two animations

As the Idle clip transitions to the Run_Left clip, the blend removes the obvious jump between poses and transitions between most body parts naturally. However, blending between the different positions of the foot results in an unnatural foot slide.

To fix foot sliding, you can manually adjust the root offset of an Animation clip so that the position of the foot changes less drastically, reducing the foot slide. To manually adjust the root offset, select the Animation clip in the Timeline Editor window. In the **Inspector** window, expand **Animation Playable Asset** and expand **Clip Root Motion Offsets**.



Select an Animation clip. In the Inspector window, expand **Animation Playable Asset** (advanced properties) and expand **Clip Root Motion Offsets**.

The **Clip Root Motion Offsets**, both position and rotation, are not zero because performing Match Offsets to Previous Clip already set these values to match the root (hips) of the previous humanoid at the end of the previous Animation clip.

Under Clip Root Motion Offsets, enable the Move tool. The Move **Gizmo** appears in the **Scene view**, at the root of the Animation clip. Use one of the following methods to manual adjust the root offset position of the Animation clip:

In the **scene** view, drag the Move Gizmo.

In the Inspector window, change the value of the appropriate Position property.

Enable the Move tool (Inspector window, blue arrow) to show the Move Gizmo (red arrow) in the Scene view. Use the Move Gizmo to manually position the root motion offset of the selected Animation clip.

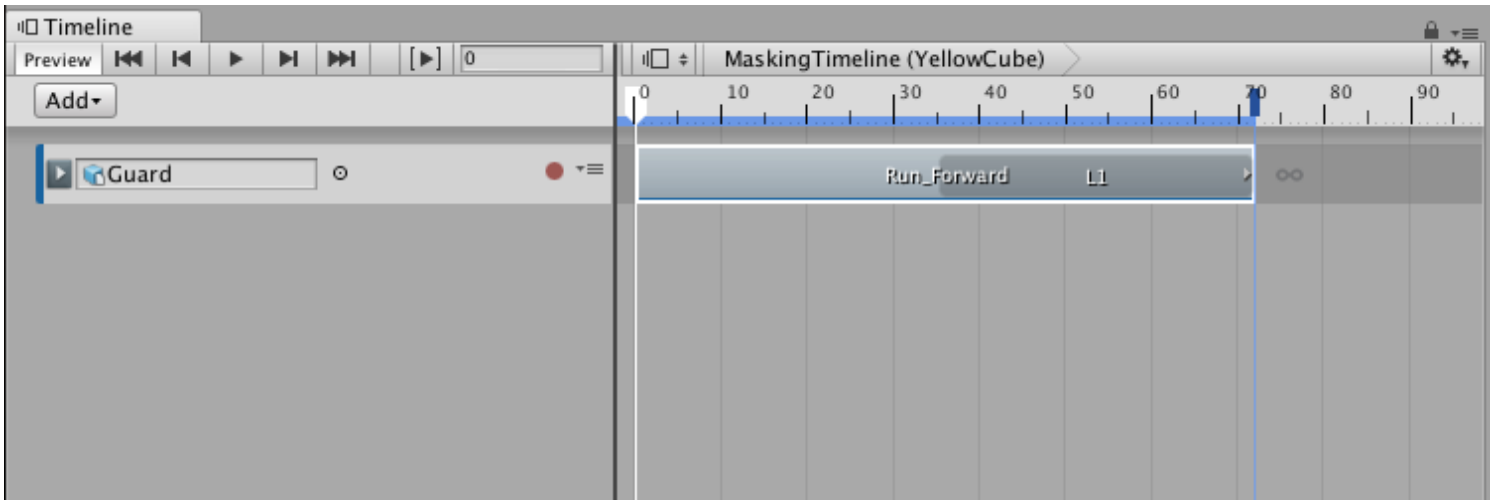
2017–12–07 Page published with limited editorial review

Using Animation Override Tracks and Avatar Masking

Leave feedback

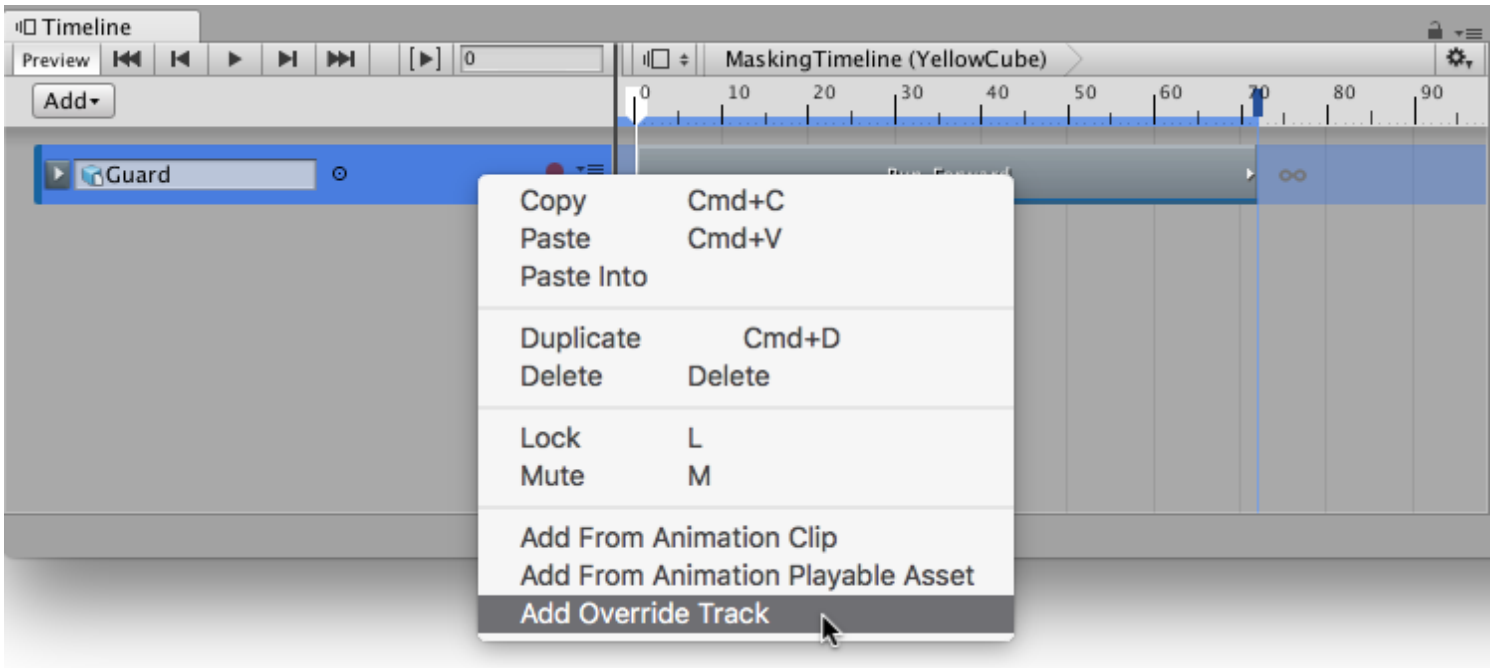
This workflow demonstrates how to use an Animation Override track and an **Avatar Mask** to replace the upper-body animation of an Animation track. Use this technique to animate a humanoid character to, for example, run and carry an object.

This workflow **does not** show how to create an Avatar mask. This workflow only demonstrates how to use an **Avatar** Mask when creating a **Timeline instance**. This workflow also assumes that you have already created a **Timeline instance** with a simple **Animation clip**, such as a walk or run cycle, on an Animation track bound to a humanoid.



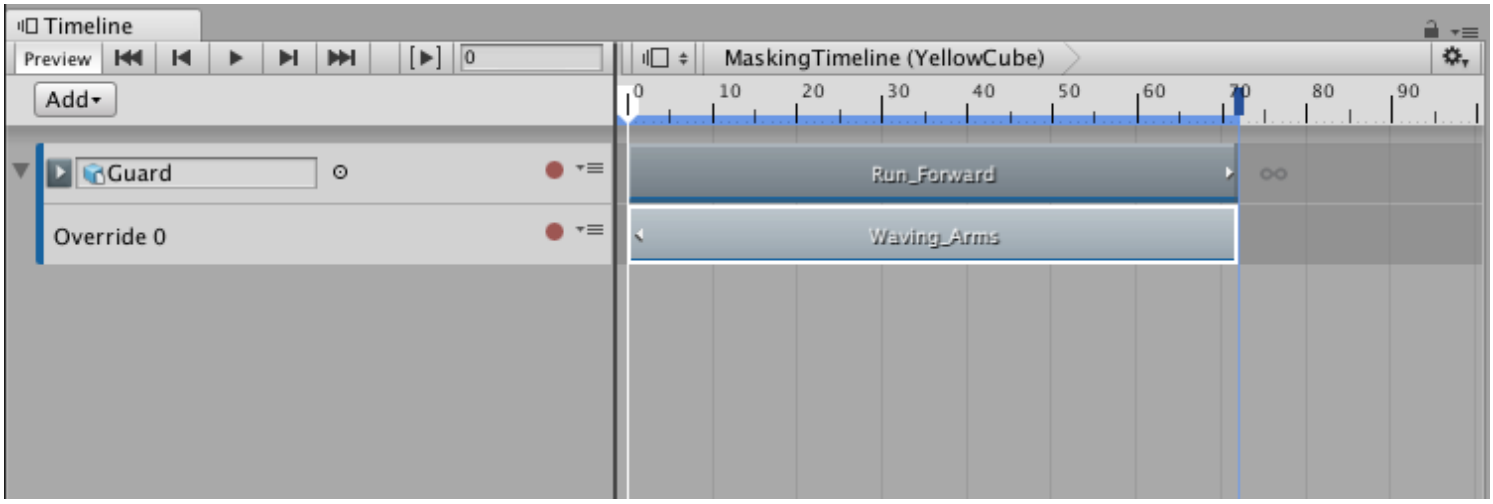
The Guard humanoid is bound to an Animation track with a run cycle (Run_Forward) that loops once

Right-click the Animation track and select Add Override Track from the context menu. An Animation Override track, named Override 0, is linked to the selected Animation track. Notice that the Animation Override track is not bound to a **GameObject**. Since the Override track is linked to the Animation track above, the Override track is bound to the same **GameObject**: the Guard humanoid.



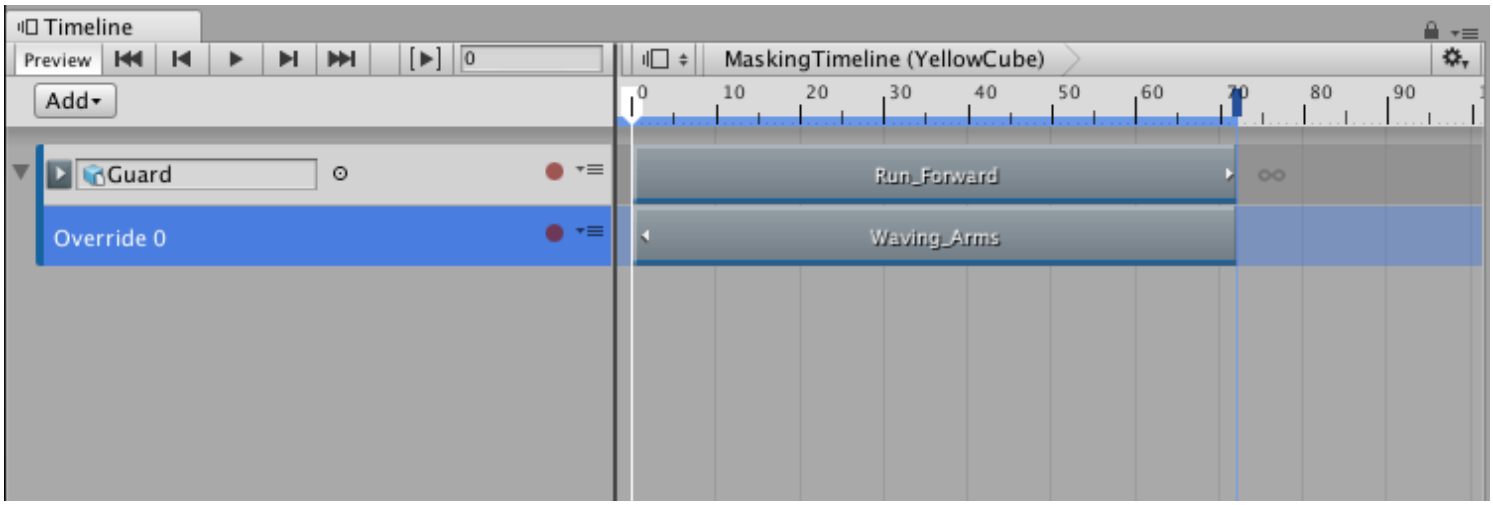
Add an Override track. Right-click the Animation track and select Add Override Track from the context menu.

From your project, drag an animation clip with upper-body animation into the Override track. For example, drag an animation of a humanoid standing still and waving their arms. Position and resize the Waving_Arms clip as appropriate.



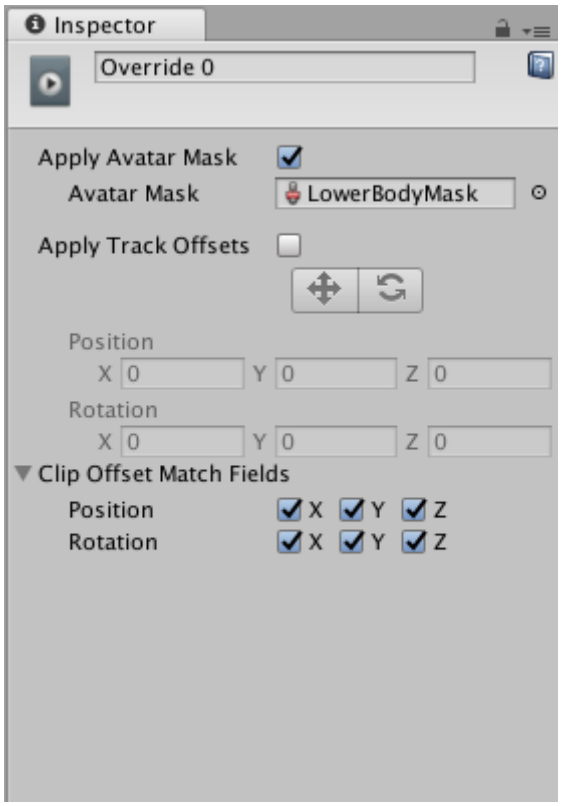
The Animation Override track contains an Animation clip of a humanoid standing still, waving their arms. The clip is resized to match the Animation clip (Run_Forward) of the parent Animation track.

Play the **Timeline** instance. The Waving_Arms clip completely overrides the Run_Forward clip. To combine the lower-body animation from the Run_Forward clip with the upper-body animation from the Waving_Arms clip, specify an Avatar Mask for the Animation Override track.

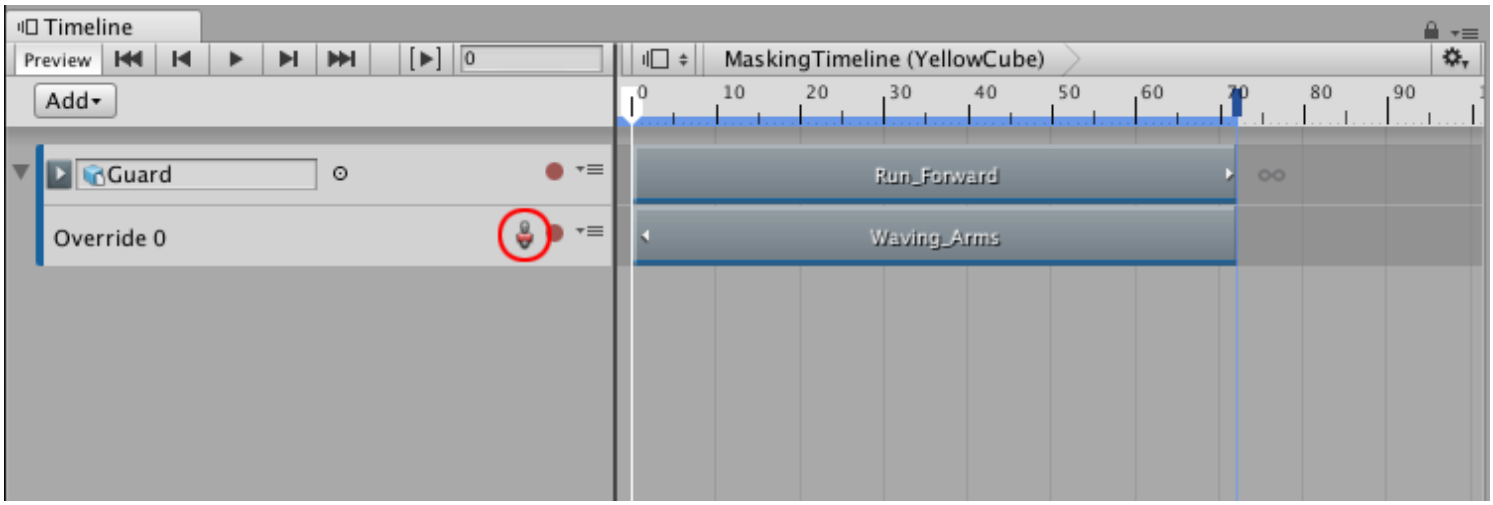


To specify an Avatar Mask, select the Override track to view its properties in the Inspector window

From the project, drag an Avatar Mask into the Avatar Mask property in the **Inspector** window. Activate the Apply Avatar Mask checkbox. An Avatar Mask icon appears beside the track name.

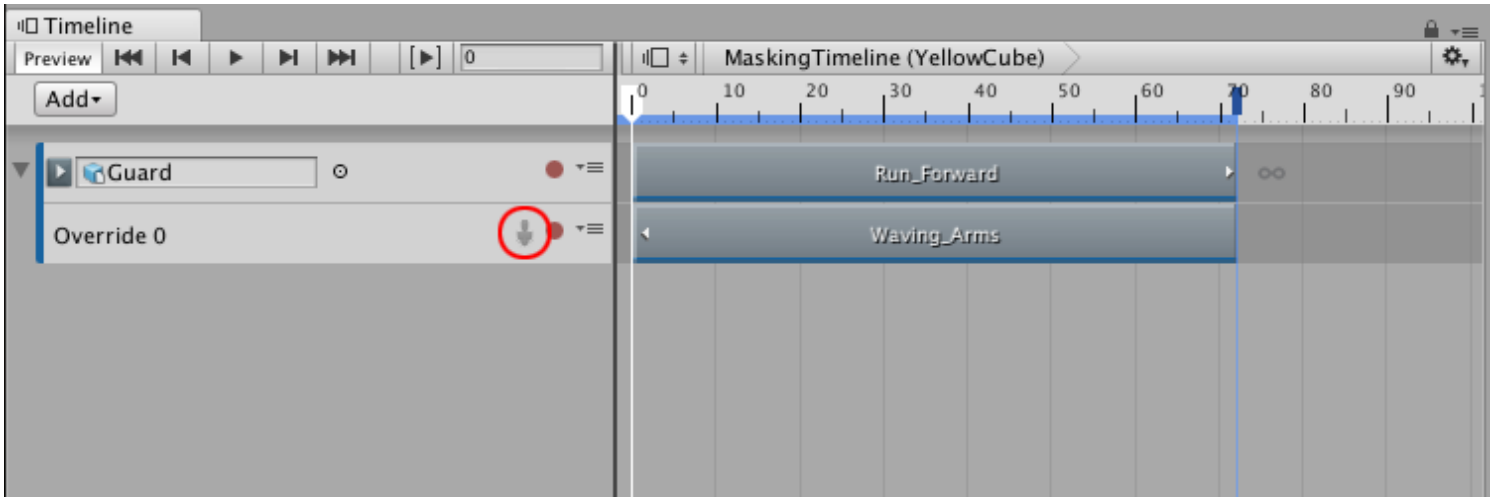


An Avatar Mask, that masks the upper-body animation, is specified for the Animation Overview clip in the Inspector window.



The Avatar Mask icon (red circle) indicates that the Animation track uses an Avatar Mask. Select and activate the Avatar Mask in the Inspector window.

Play the Timeline instance. On the Guard humanoid, the upper-body animation is taken from the Waving_Arms clip and the lower-body animation is from the Run_Forward clip. To temporarily disable the Avatar Mask, click the Avatar Mask icon.



The Avatar Mask icon (red circle) disappears when disabled. The Waving_Arms animation completely overrides the Run_Forward animation.

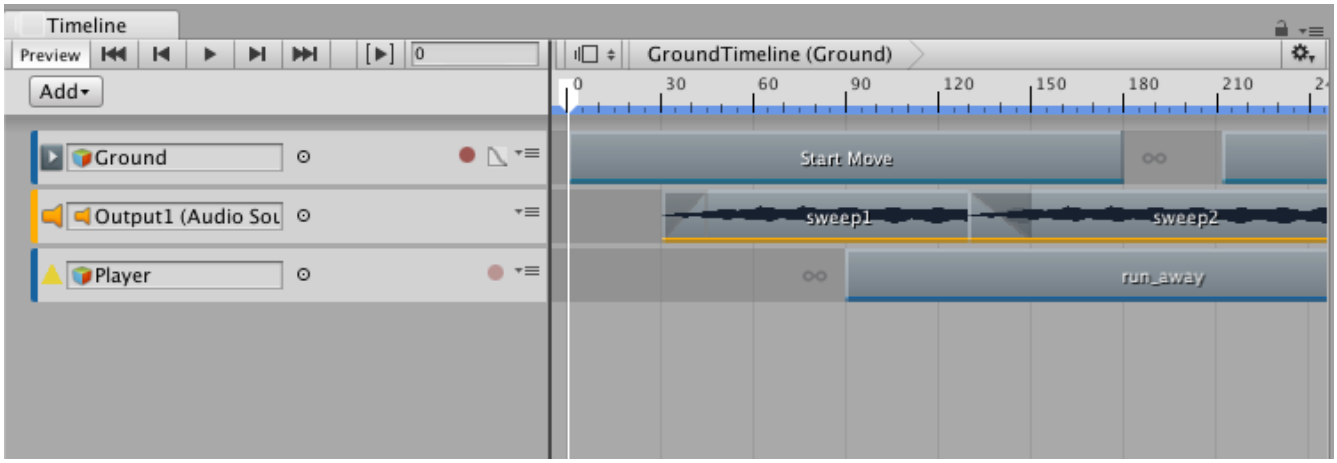
2017–12–07 Page published with limited editorial review

Timeline Editor window

Leave feedback

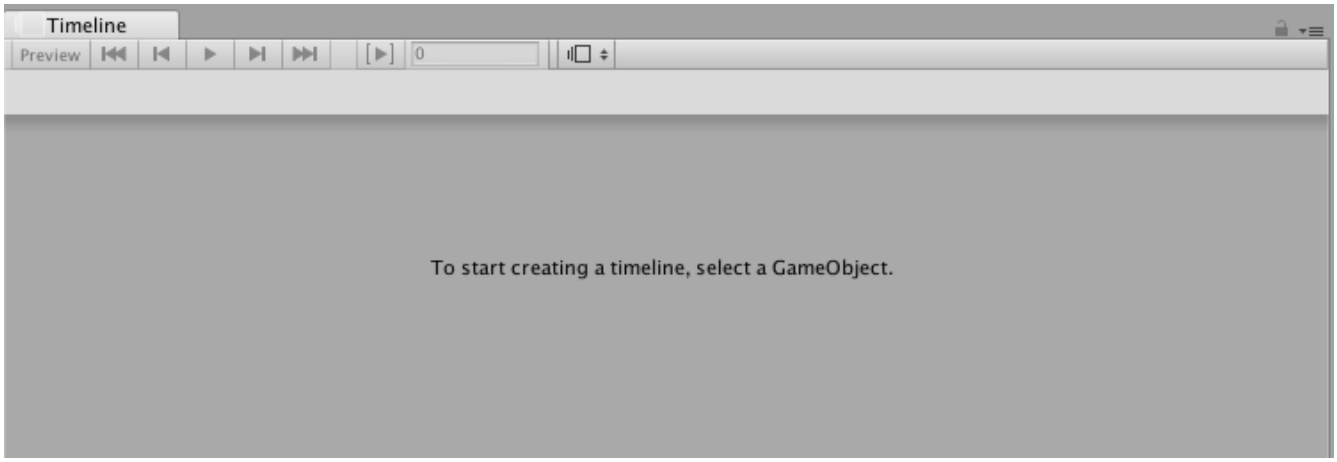
To access the **Timeline** Editor window, select **Timeline Editor** from the Window menu. What the **Timeline** Editor window shows depends on what you select in either the Project window or the **Scene view**.

For example, If you select a **GameObject** that is associated with a **Timeline Asset**, the **Timeline Editor window** shows the tracks and clips from the **Timeline Asset** and the **GameObject** bindings from the **Timeline instance**.



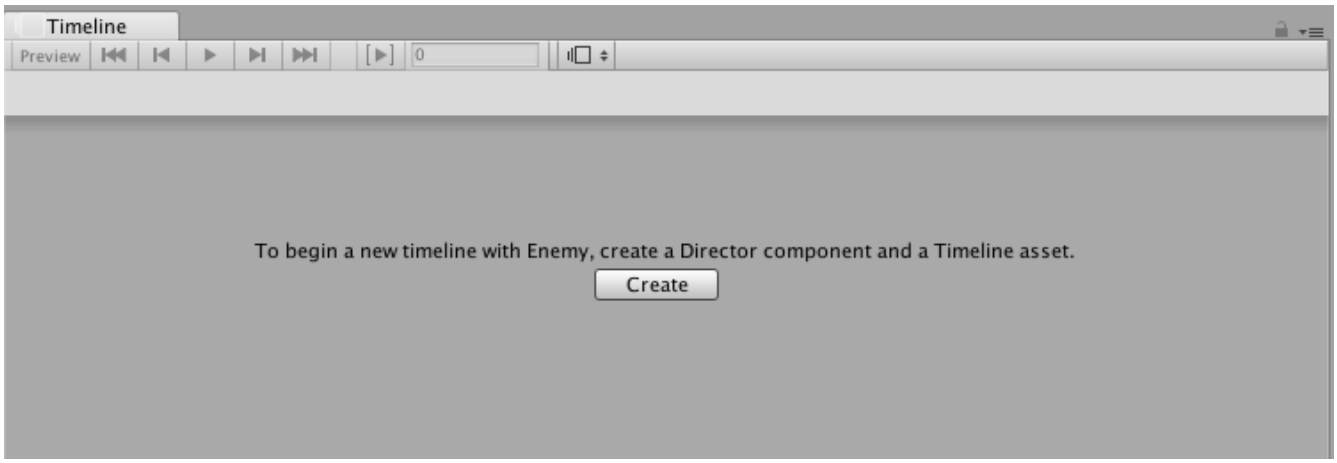
Selecting a GameObject associated with a Timeline Asset displays its tracks, its clips, and the bindings from the TImeline instance

If you haven't selected a GameObject, the Timeline Editor window informs you that the first step for creating a Timeline **Asset** and a Timeline instance is to select a GameObject.



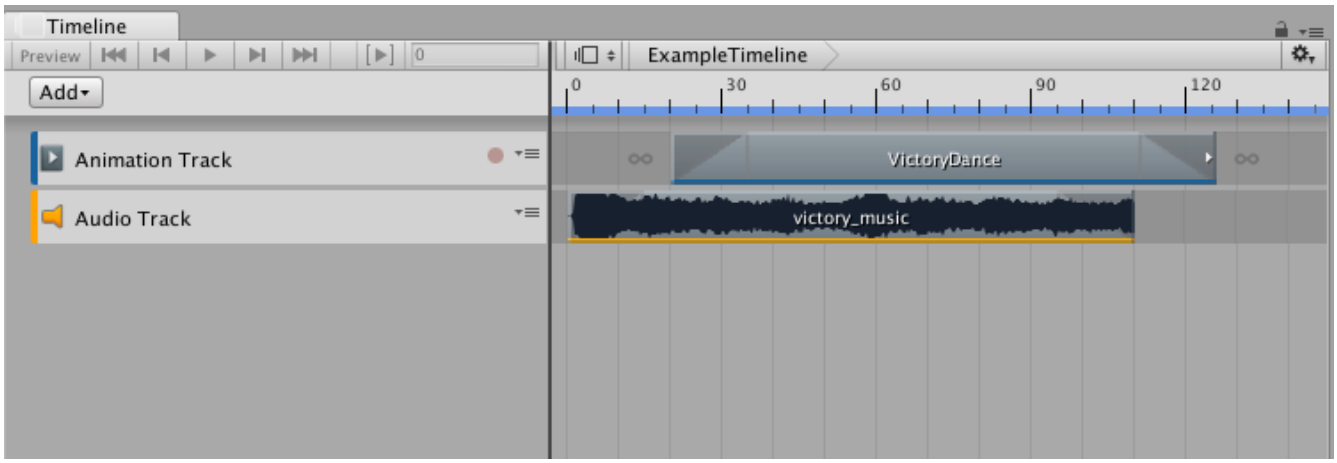
With no GameObject selected, the Timeline Editor window provides instructions

If a GameObject is selected and it is not associated with a Timeline Asset, the Timeline Editor window provides the option for creating a new Timeline Asset, adding the necessary **components** to the selected GameObject, and creating a Timeline instance.



Select a GameObject that is not associated with a Timeline Asset to create a new Timeline Asset, add components, and create a Timeline instance.

To use the Timeline Editor window to view a previously created Timeline Asset, select the Timeline Asset in the Project window and open the Timeline Editor window. The Timeline Editor window shows the tracks and clips associated with the Timeline Asset, but without the track bindings to GameObjects in the **scene**. In addition, the **Timeline Playback Controls** are disabled and there is no **Timeline Playhead**.



Timeline Asset selected in the Project window shows its tracks and clips but with no track bindings, Timeline Playback Controls, or Timeline Playhead

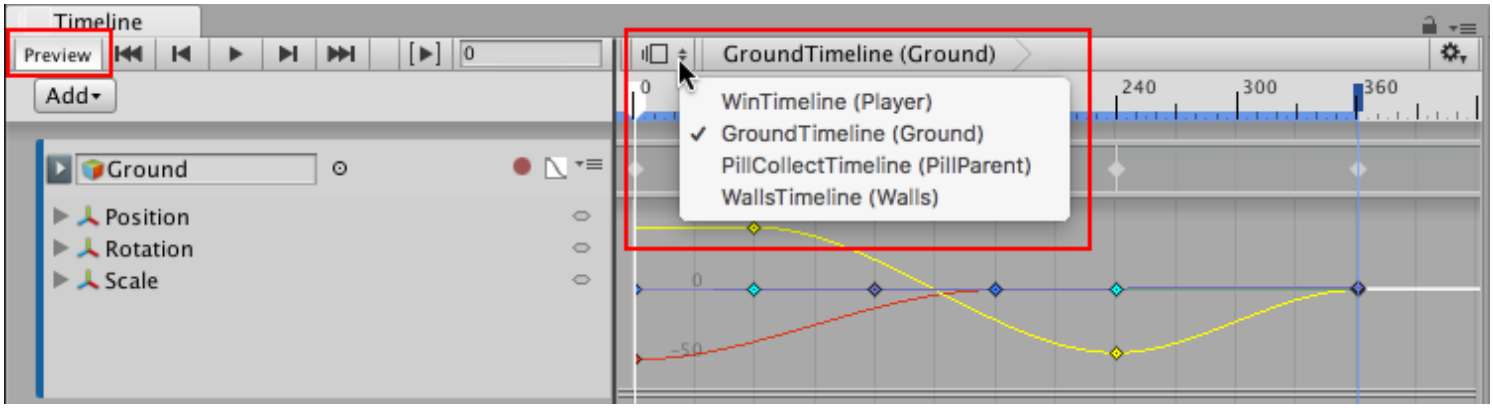
The track bindings to GameObjects in the scene are not saved with the Timeline Asset. The track bindings are saved with the Timeline instance. For details on the relationship between the project, the scene, Timeline Assets, and Timeline instances, see Timeline overview.

2017–08–10 Page published with limited editorial review

Timeline Preview and Timeline Selector

Leave feedback

Use the **Timeline** Selector to select the **Timeline instance** to view, modify, or preview in the **Timeline** Editor window. The **Timeline** Preview button enables or disables previewing the effect the selected **Timeline instance** has on your **scene**.



Timeline Preview button with Timeline Selector and menu. Selecting a Timeline instance automatically enables the Timeline Preview button.

To select a Timeline instance, click the **Timeline Selector** to show the list of Timeline instances in the current scene.

Each menu item displays the name of the **Timeline Asset** and its associated **GameObject** in the current scene. For example, the **Timeline Asset** named GroundTimeline that is associated with the Ground **GameObject**, displays as “GroundTimeline (Ground).”

2017–08–10 Page published with limited editorial review

Timeline Playback Controls

Leave feedback

Use the buttons and fields in the **Timeline** Playback Controls to play the **Timeline instance** and to control the location of the **Timeline Playhead**.

Timeline Playback Controls

Timeline Start button

Click the Timeline Start button, or hold Shift and press Comma (,), to move the Timeline Playhead to the start of the Timeline instance.

Previous Frame button

Click the Previous Frame button, or press Comma (,), to move the Timeline Playhead to the previous frame.

Timeline Play button

Click the Timeline Play button, or press the Spacebar, to preview the Timeline instance in **Timeline Playback mode**. **Timeline Playback mode** does the following:

Begins playback at the current location of the Timeline Playhead and continues to the end of the Timeline instance. If the Play Range button is enabled, playback is restricted to a specified time range.

The Timeline Playhead position moves along the Timeline instance. The **Playhead Location field** shows the position of the Timeline Playhead in either frames or seconds, depending on the Timeline settings.

To pause playback, click the Timeline Play button again, or press the Spacebar.

When playback reaches the end of the Timeline instance, the Wrap Mode determines whether playback should hold, repeat, or do nothing. The Wrap Mode setting is a property of the the Playable Director component.

Timeline Playback mode provides a preview of the Timeline instance while in the **Timeline Editor window**. Timeline Playback mode is only a simulation of Game Mode that does not support audio playback. To preview a Timeline instance with audio, you can enable the **Play on Awake** option in the Playback Director **component** and preview game play by using the Play mode.

Next Frame button

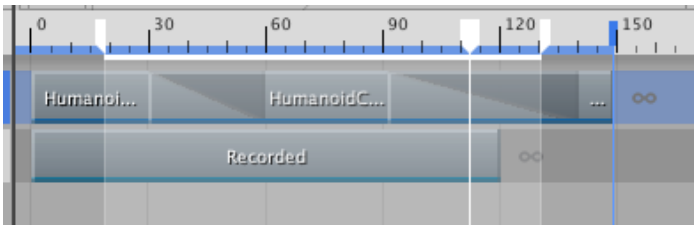
Click the Next Frame button, or press Period (.), to move the Timeline Playhead to the next frame.

Timeline End button

Click the Timeline End button, or hold Shift and press Period (,), to move the Timeline Playhead to the end of the timeline.

Play Range button

Enable the Play Range button to restrict playback to a specific range of seconds or frames. The timeline highlights the play range and indicates its start and end with white markers. To modify the play range, drag either marker.



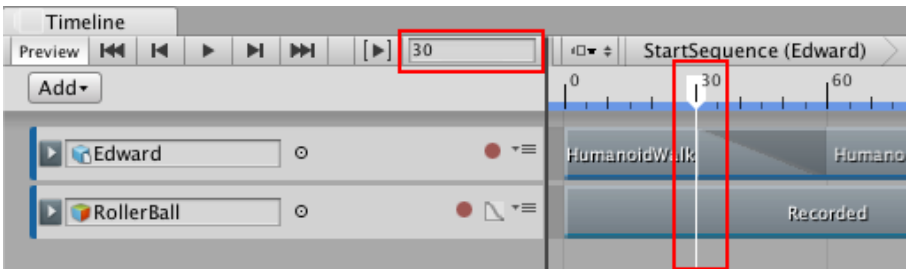
Play Range enabled with while markers defining range

The **Wrap Mode** property of the Playable Director component determines what happens when playback reaches the end marker.

You can only set a play range when previewing a Timeline instance within the Timeline Editor window. Unity ignores the play range in Play mode.

The Playhead Location field and Timeline Playhead

The Timeline Playhead indicates the exact point in time being previewed in the Timeline Editor window. The Playhead Location field expresses the location of the Timeline Playhead in either frames or seconds.



Playhead Location field and Timeline Playhead

To jump the Timeline Playhead to a specific time, click the timeline. You can also enter the time value in the Playhead Location field and press Enter. When typing a value, frames are converted to seconds or seconds are converted to frames, based on the Timeline settings. For example, if the timeline is expressed as seconds with a frame rate of 30 **frames per second**, entering 180 in the Playhead Location field converts 180 frames to seconds and moves the Timeline Playhead to 6:00.

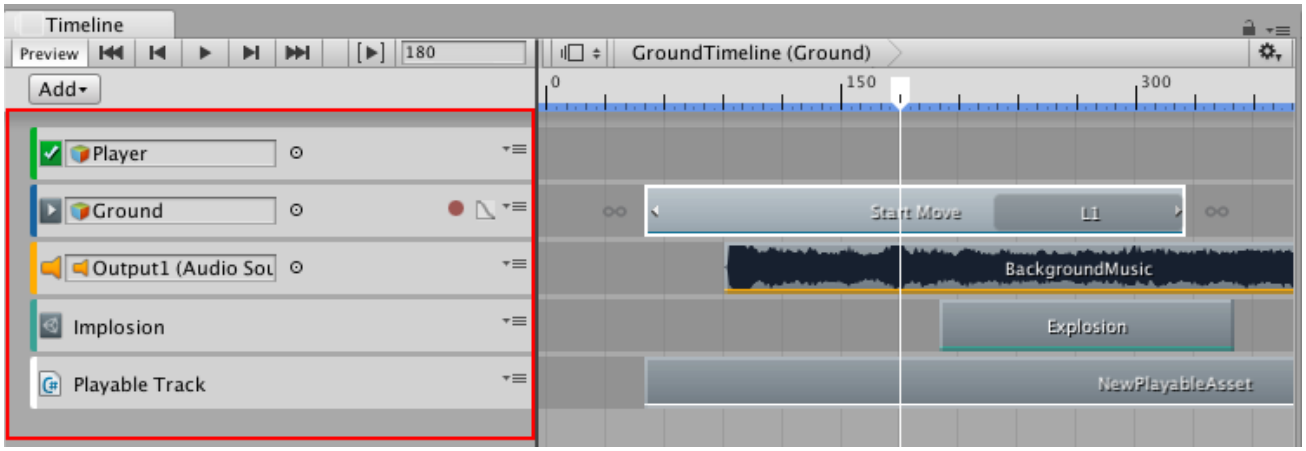
To set the time format used by the Timeline Editor window, use the Timeline Settings.

2017–08–10 Page published with limited editorial review

Track List

Leave feedback

Use the Track List to add, select, duplicate, delete, lock, mute, and reorder the tracks that comprise a **Timeline Asset**. You can also organize tracks into **Track groups**.



Track list

A colored accent line identifies each type of track. By default, Activation tracks are green, Animation tracks are blue, Audio tracks are orange, Control tracks are turquoise, and Playable tracks are white.

The bindings on each track are saved to the **Timeline instance**. Playable Director **component** associated with the **GameObject** linked to the **Timeline Asset**. This association is referred to as a **Timeline instance**. See Timeline overview for details.

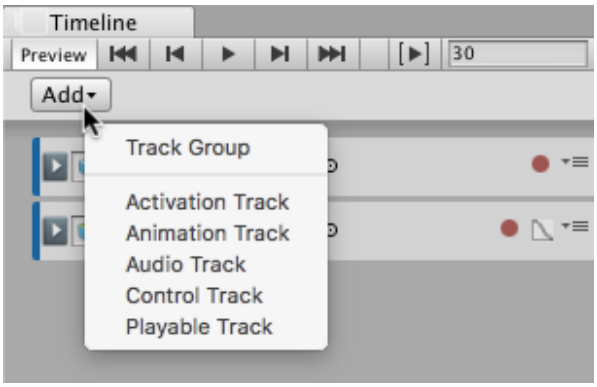
2017–08–10 Page published with limited editorial review

Adding tracks

Leave feedback

The **Timeline** Editor window supports many different methods of adding tracks to the **Track list**. The method you choose impacts **GameObjects**, Track binding, and **components**.

The simplest method of adding a track is to click the Add button and select the type of track from the Add Track menu. You can also Right-click in an empty area of the Track list and select the type of track from the Add Track menu.



Add Track menu

The **Timeline Editor window** also supports dragging a GameObject into the Track list. Drag a GameObject into an empty area in the Track list and select the type of track to add from the context menu. Depending on the type of track selected, the **Timeline Editor window** performs different actions:

Select Animation Track and the Timeline Editor binds the GameObject to the Animation track. If the GameObject doesn't already have an **Animator component**, the Timeline Editor creates an **Animator component** for the GameObject.

Select Activation Track and the Timeline Editor binds the GameObject to the Activation track. There are some limitations when creating an Activation track when dragging a GameObject. For example, the main GameObject with the Playable Directory component should not be bound to an Activation track. Since this is the same GameObject that links the **Timeline Asset** to the **scene**, activating and disabling the GameObject affects the length of **Timeline instance**.

Select Audio Track and the Timeline Editor adds an **Audio Source** component to the GameObject and binds this **Audio Source** component to the Audio Track.

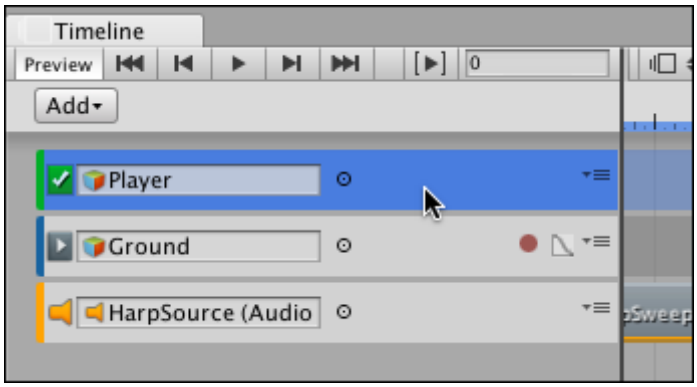
2017–08–10 Page published with limited editorial review

Selecting tracks

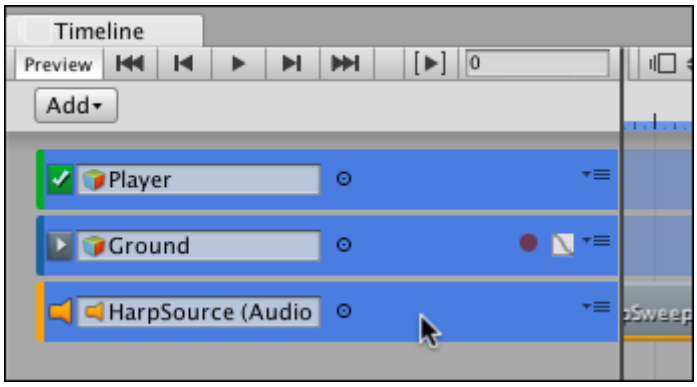
Leave feedback

Click to select a single track. Selecting a track deselects all other tracks or clips. Selecting a track shows its properties in the **Inspector** window. The properties available change depending on the type of track selected.

To select contiguous tracks, select the first track and then hold Shift and click the last track in the series. For example, to select three contiguous tracks, click the first track, then hold Shift and click the third track. All three tracks are selected.



Click to select the first track



Hold Shift and click to select contiguous tracks

Hold Command/Control and click to select discontiguous tracks. Hold Command/Control and click to deselect a selected track.

2017–08–10 Page published with limited editorial review

Duplicating tracks

Leave feedback

The **Timeline** Editor window supports the following different methods of duplicating tracks:

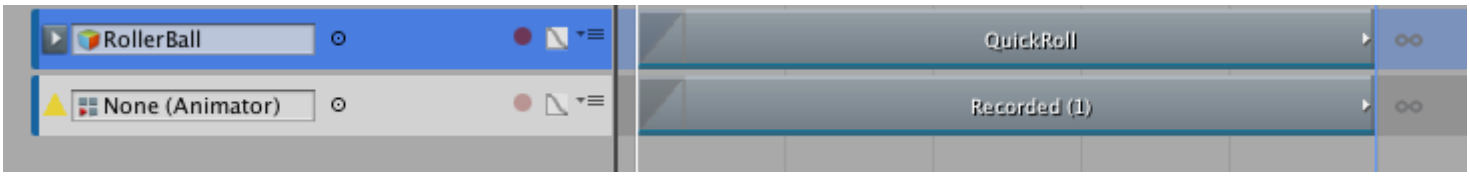
Select a track. Right-click in an empty area in the **Track list** and select **Duplicate** from the context menu.

Select a track. Hold Command/Control and press D.

Select a track. Hold Command/Control and press C. for copy, and press V for paste.

Right-click a track and either select **Duplicate** from the context menu or hold Command/Control and press D.

Duplicating a track copies its clips, blends, and **Inspector** properties. If the duplicated track is bound to a **GameObject**, the binding is reset to None.



Track binding for the duplicated track is reset to None

2017–08–10 Page published with limited editorial review

Deleting tracks

Leave feedback

Delete a track to remove the track, its clips, blends, and properties from the **Timeline** Editor window. This is a destructive action that modifies a **Timeline Asset** and affects all **Timeline** instances based on the **Timeline Asset**. To delete a track, Right-click the track and select **Delete** from the context menu.

Deleting an Animation track also deletes the recorded **Infinite clips** for **Animation clips** that were converted from **Infinite clips**.

The Project window may still show Infinite clips as children of a **Timeline Asset** since it is not updated until after the **scene** or project is saved.

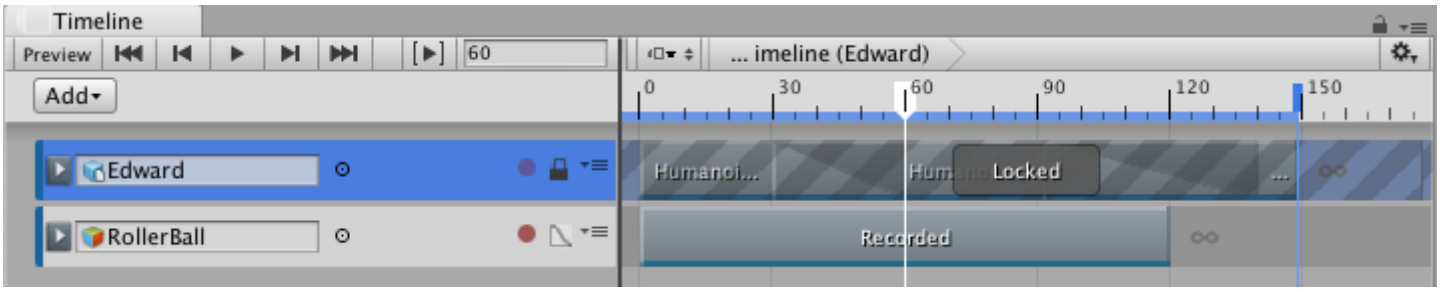
2017–08–10 Page published with limited editorial review

Locking tracks

Leave feedback

Lock a track to lock editing of the track and any of the clips used by the track.

Use lock when the animation on a track is completed and you want to avoid inadvertently modifying the track. A locked track cannot be edited and its clips cannot be selected. The Lock icon identifies a locked track.



Selected and locked track

To lock a track, Right-click on the track and select **Lock** from the context menu. You can also select a track and press L. You can select and lock multiple tracks at a time.

Note: Locked tracks can still be deleted.

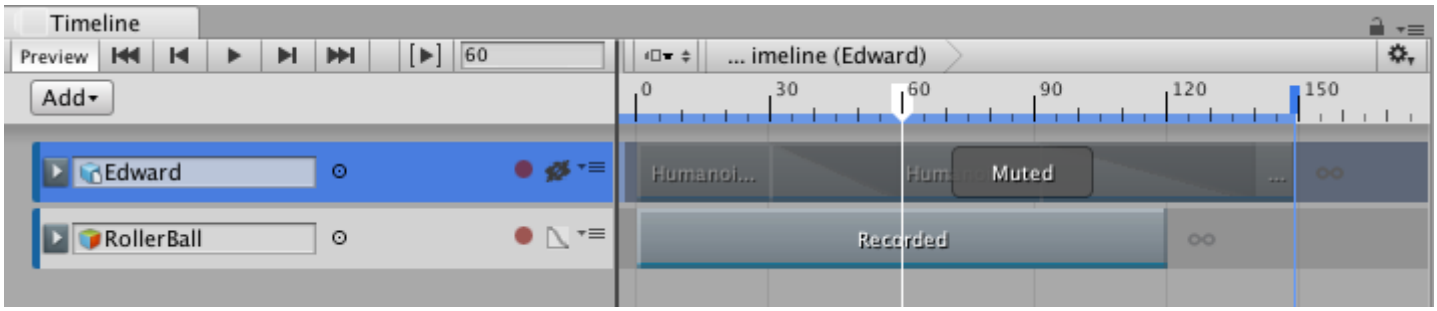
2017–08–10 Page published with limited editorial review

Muting tracks

Leave feedback

Mute a track to disable its clips and hide them from the **scene**.

You can also use mute when your **Timeline instance** includes many tracks with animations and you want to focus on the animation of one or a few tracks. The Mute icon identifies a muted track.



Selected and muted track

To mute a track, Right-click on the track and select **Mute** from the context menu. You can also select a track and press M. You can select and mute multiple tracks at a time. To unmute a track, click the Mute icon.

2017–08–10 Page published with limited editorial review

Reordering tracks and rendering priority

Leave feedback

The **rendering** and animation priority of tracks is from top to bottom. Reorder tracks to change their **rendering** or animation priority.

For example, if a **Track list** contains two Animation tracks that animate the position of a **GameObject**, the second track overrides the animation on the first track. The animation priority is the reason why Animation Override tracks are added as child tracks under Animation tracks.

To reorder tracks, select one or more tracks and drag until a white insert line appears between tracks in the Track list. The white insert line indicates the destination of the tracks being dragged. Release the mouse button to reorder tracks.

An Animation Override track is bound to the same GameObject as its parent Animation track. Reordering an Animation Override track converts it to an Animation track and resets its binding to none.

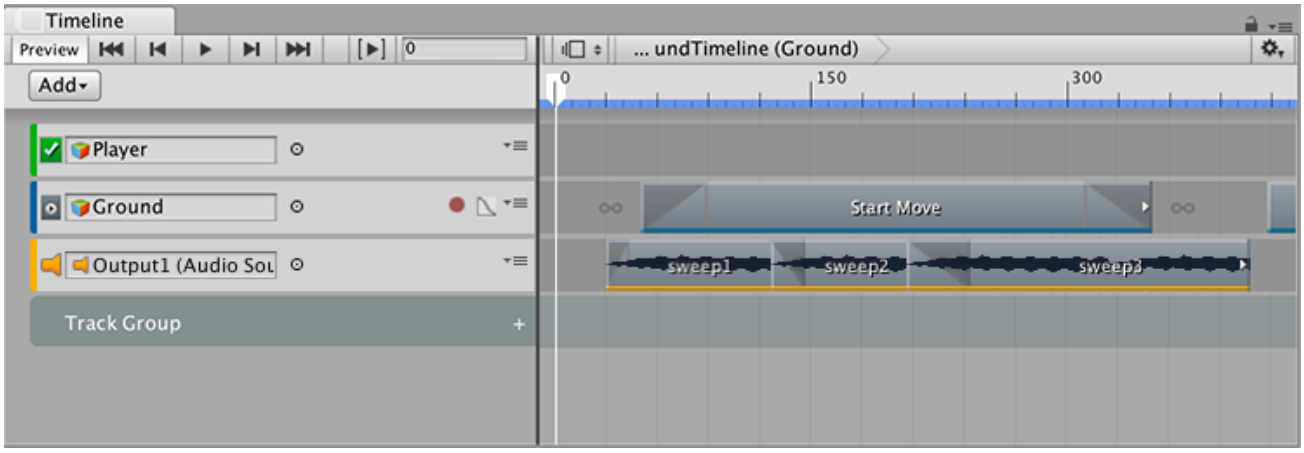
2017–12–07 Page amended with limited editorial review

Organizing tracks into Track groups

Leave feedback

Use Track groups to organize tracks when you are working with many tracks. For example, a **Timeline Asset** contains an Animation track and an Audio track that interact with the same **GameObject**. To organize these tracks, you can move these tracks into their own Track group.

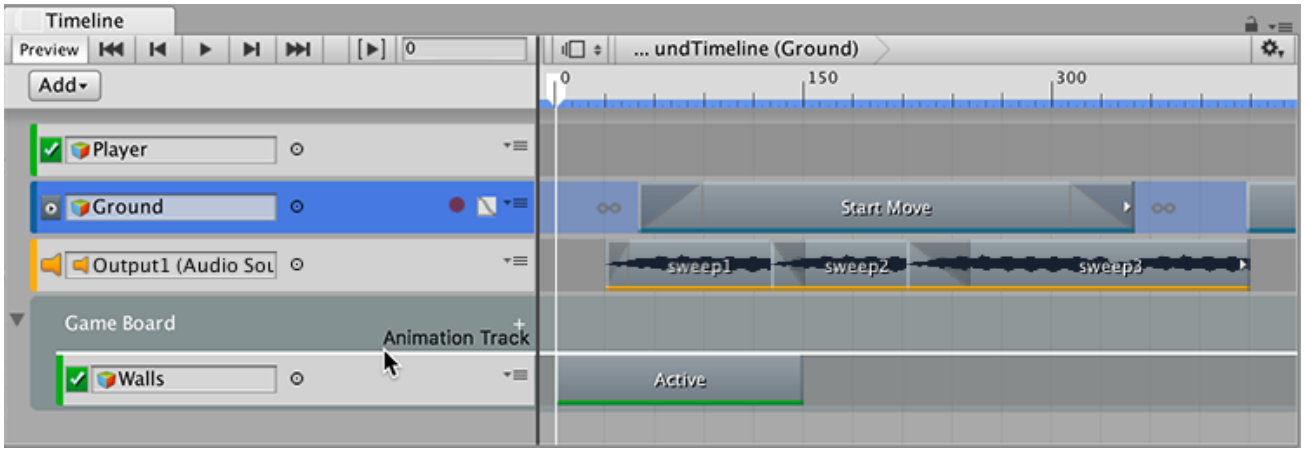
To add a Track group, click the Add button and select **Track Group** from the Add menu. You can also Right-click an empty area of the **Track list** and select **Track Group** from the context menu. A new Track group is added at the bottom of the **Track list**.



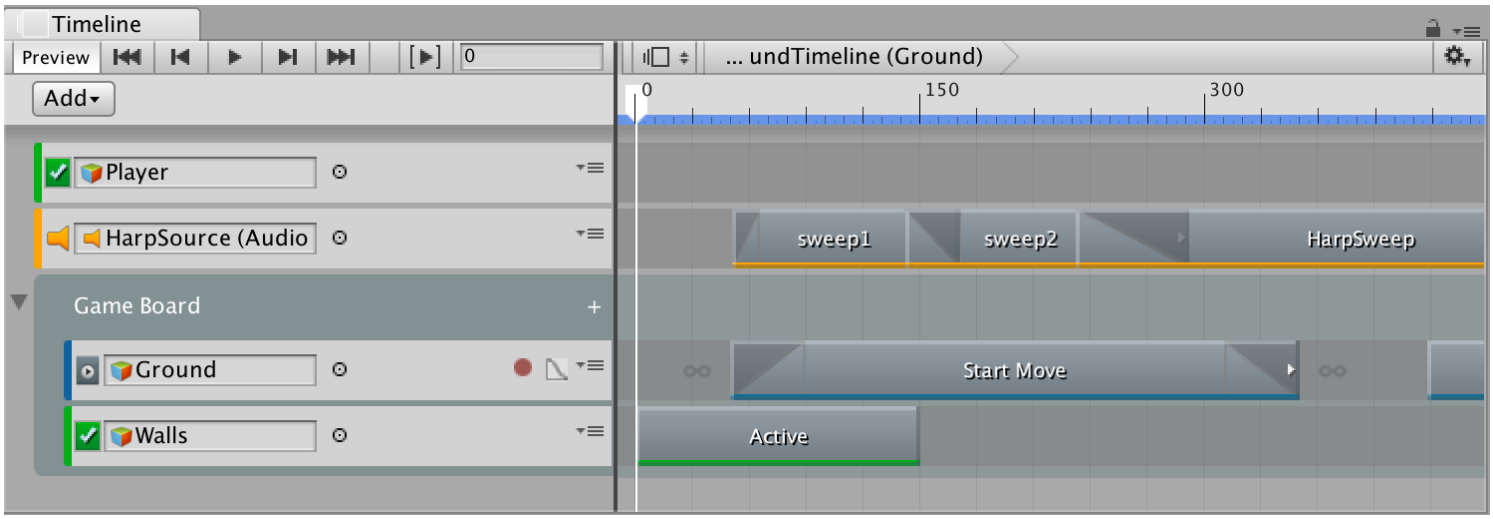
Timeline Editor window with Track group added

To rename a Track group, click the name and an I-beam cursor appears. Type the new name for the Track group and press Return.

To move tracks into a Track group, select one or more tracks and drag over the Track group. The Track group is highlighted. When dragging a selection of tracks, the last selected track type displays beside the cursor. To drop the tracks before a specific track in the Track group, drag until a white insert line indicates the destination.

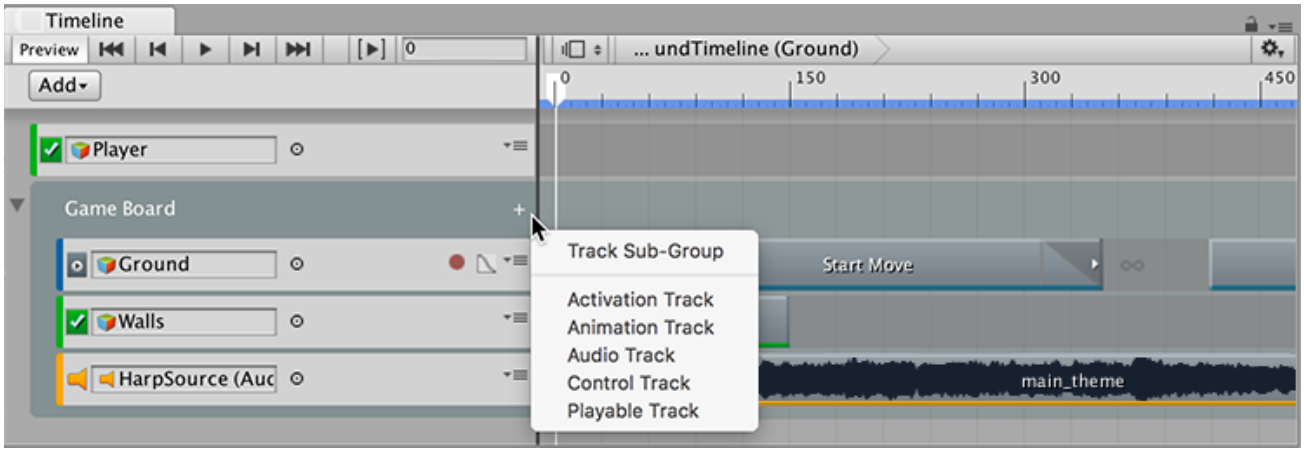


Release the mouse button when the white insert line appears within the Track group



Selected tracks are moved to the location of the insert line

A Track group can also have any number of Track sub-groups. To add a Track sub-group, either select a Track group and click the Add button in the Track list, or click the Plus icon beside the Track group name, and select **Track Sub-Group**. You can also use this menu to add tracks directly to a Track group or a Track sub-group.



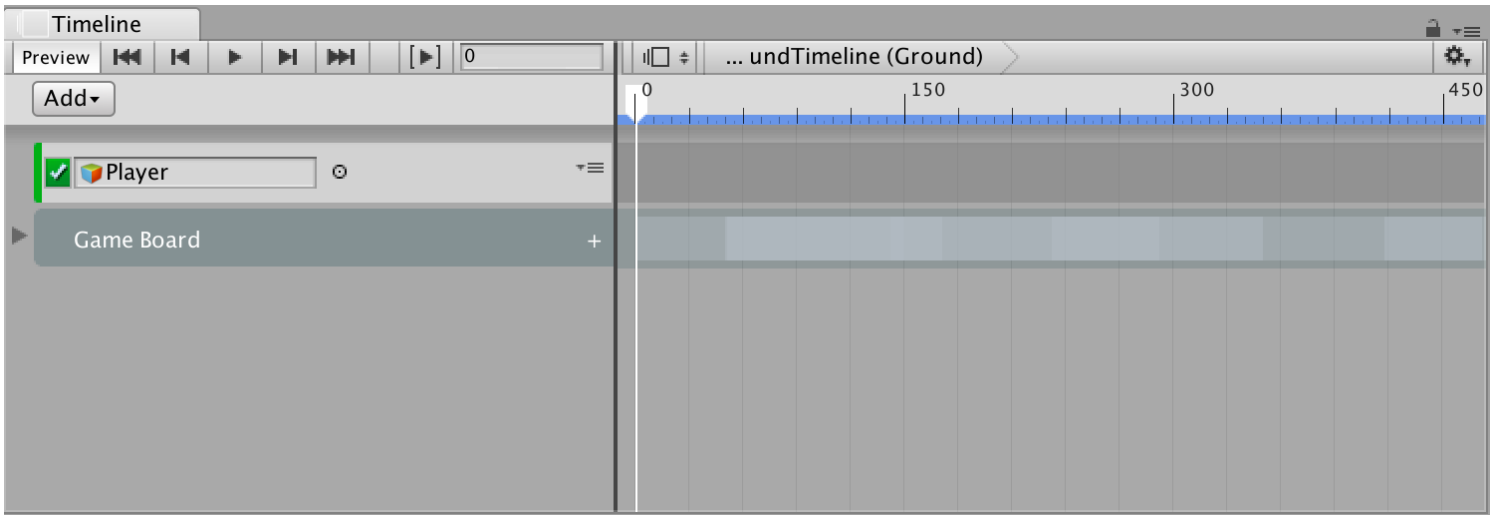
Click the Plus icon to add tracks and sub-groups to Track groups

2017–08–10 Page published with limited editorial review

Hiding and showing Track groups

Leave feedback

To hide the tracks in a **Track group**, click the Triangle icon beside the name of the **Track group**. The tracks are not muted; the tracks are hidden from view in the **Timeline** Editor window. To show the tracks in a **Track group**, click the Triangle icon again.



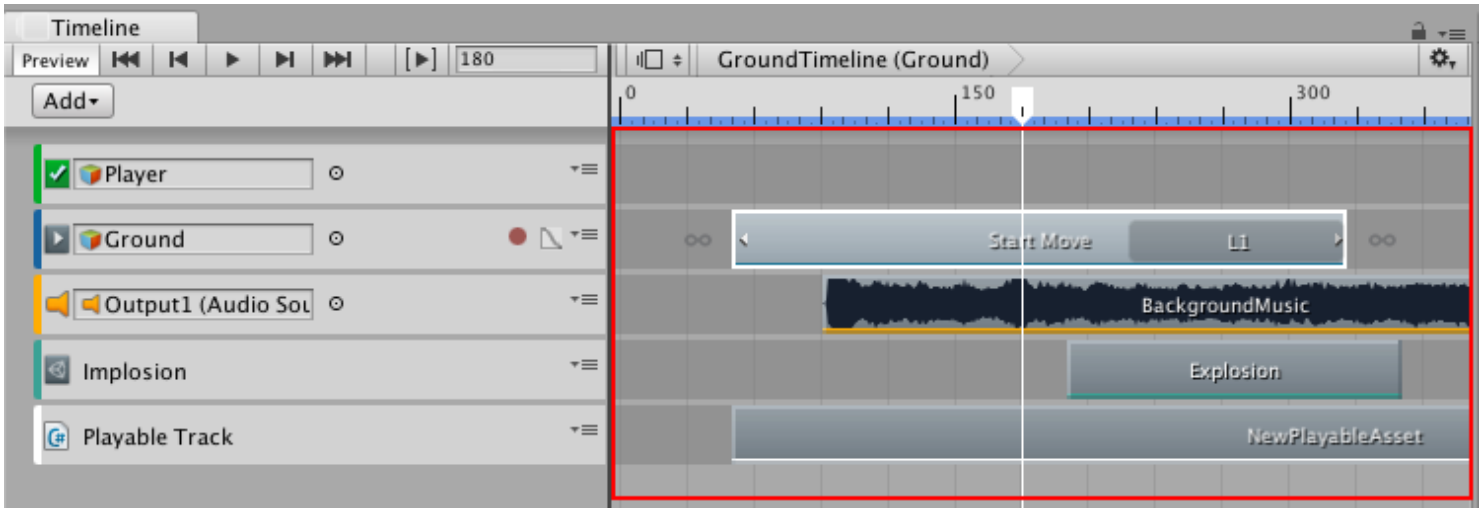
Triangle icon hides the tracks in the Game Board Track group. A ghost track visually represents the hidden tracks.

2017–08–10 Page published with limited editorial review

Clips view

Leave feedback

The **Clips view** is where you add, position, and manipulate clips for each track in the **Track list**. A clip cannot exist in the **Timeline** Editor window without a track.



Clips view shows the clips for each track

Each clip has a colored accent line to identify the type of track and clip. By default, Activation clips are green, **Animation clips** are blue, **Audio clips** are orange, Control clips are turquoise, and Playable clips are white.

2017–08–10 Page published with limited editorial review

Navigating the Clips view

Leave feedback

Use one of the following methods to pan, zoom, or frame clips in the **Clips view**:

To pan, Middle-drag, or hold Alt and drag.

To zoom vertically, move the scroll-wheel, or hold Alt and right-drag.

To zoom horizontally, hold Command/Control and zoom vertically.

To frame all selected clips, select one or multiple clips and press F.

To frame all clips vertically, press A.

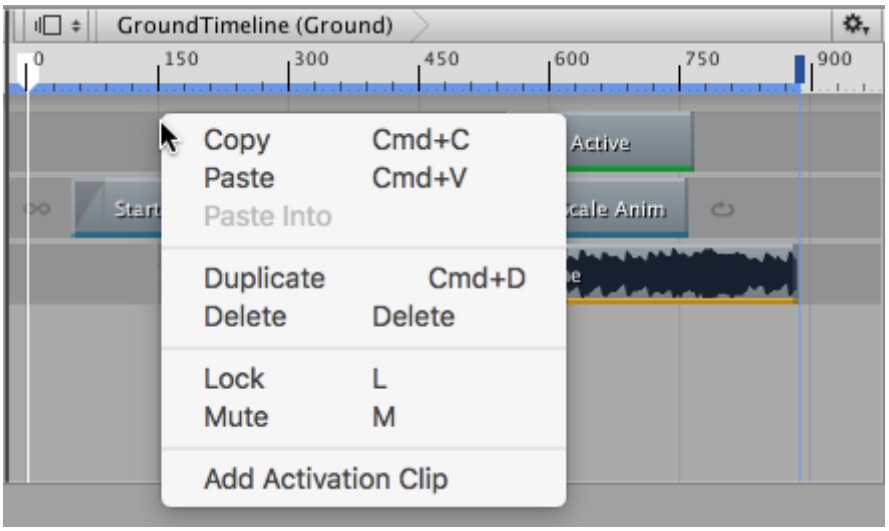
2017–08–10 Page published with limited editorial review

Adding clips

Leave feedback

The **Timeline** Editor window supports different methods of adding clips to tracks depending on the type of track.

The quickest method is to Right-click on an empty area within a track and select the appropriate Add option from the context menu. Depending on the track, the options for adding clips change. The clip is added after the last clip on the track.



Context menu for adding Activation clips

You can also drag an **Animation clip** to an empty area in the **Timeline Editor window** to automatically create a track and add the **Animation clip** to the track.

2017–08–10 Page published with limited editorial review

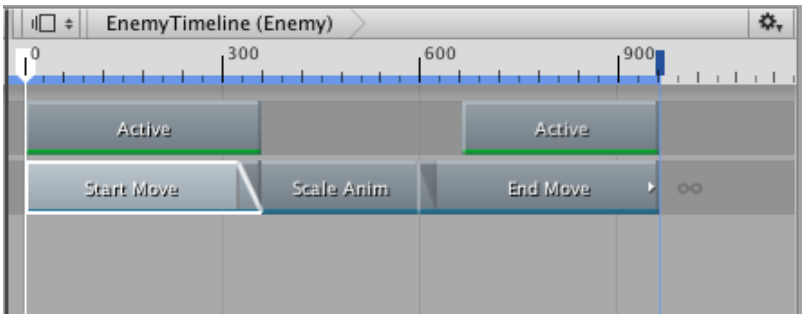
Selecting clips

Leave feedback

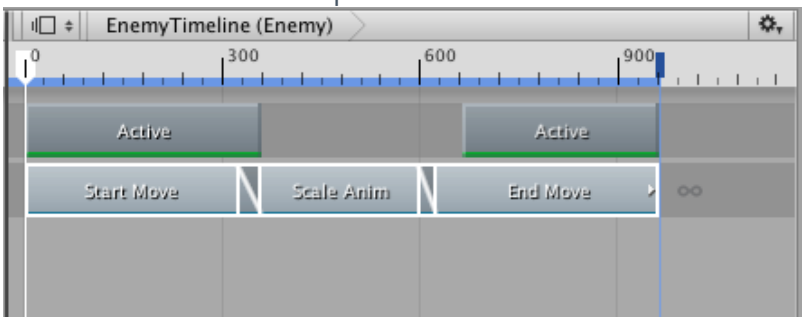
Click to select a single clip. Selecting a clip deselects all other tracks or clips. The Clip area displays the selected clip with a white border.

Selecting a clip shows its properties in the **Inspector** window, allowing you to change the start of the clip, its duration, and other clip properties. The properties available change depending on the type of clip selected. See Timeline Inspector for details.

Hold Shift and click to select contiguous clips. For example, to select three contiguous clips, click the first clip, then hold Shift and click the third clip. All three clips are selected.



Click to select the first clip

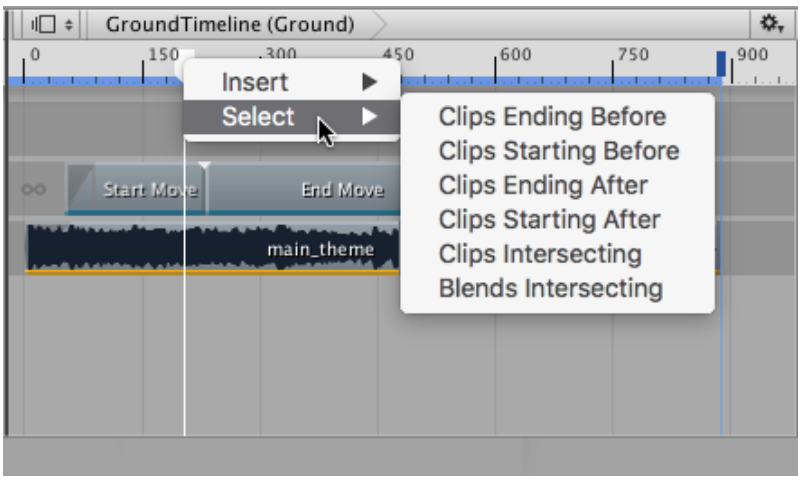


Shift-click the third clip to select contiguous clips

Hold Command/Control and click to select discontiguous clips. Hold Command/Control and click a selected clip to deselect it.

Click and drag on an empty area in the **Clips view** to draw a selection rectangle. This selects all clips that intersect the rectangle. Hold down Shift while drawing the selection rectangle to add clips to the current selection.

You can also select clips with the **Timeline Playhead**. Right-click the **Timeline Playhead** on the **timeline** above the Clips view and choose a selection option. This selects clips that either start after, start before, end after, end before, or intersect the **Timeline Playhead**. Clips are selected on all tracks.



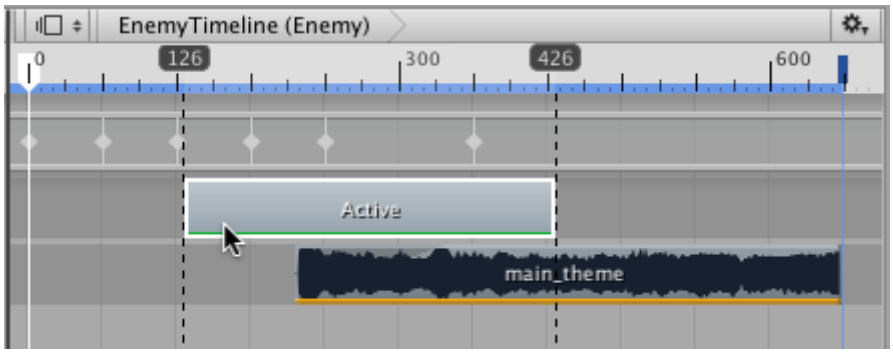
Right-click the Timeline Playhead for more clip selection options

2017–08–10 Page published with limited editorial review

Positioning clips

Leave feedback

To position clips on a track, select one or more clips and drag. While dragging, black guides indicate the range of clips being positioned. The **timeline** shows the start time and end time of the clips being positioned.



Drag to position one or more selected clips

You can also position a clip by selecting the clip and changing its start time in the Inspector window. This method only works when selecting a single clip. The **Inspector** window does not show properties for multiple clips.

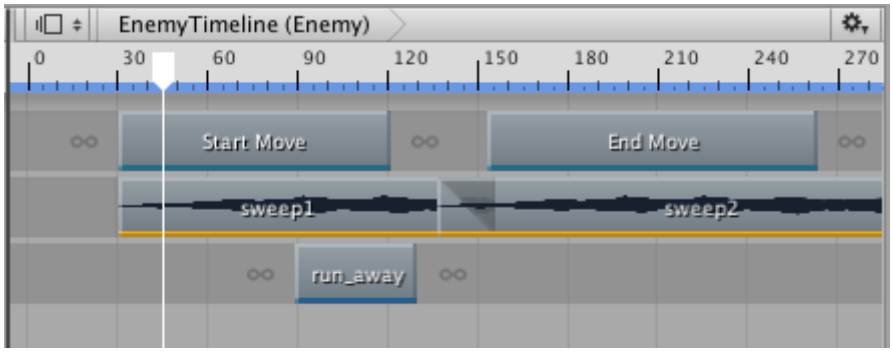
You can move a clip to another track of the same type. Drag the selection vertically and a ghost of the selected clips helps visualize the result of moving the clip.

While positioning clips, if a clip overlaps another clip on the same track, either a blend or an override occurs, depending on the type of track:

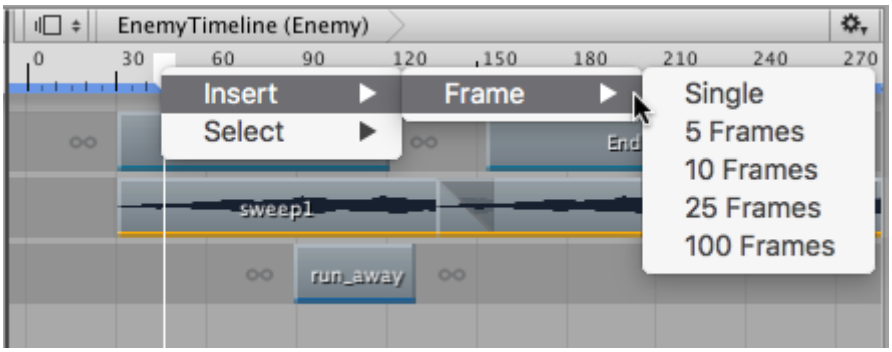
Activation track, Control track, or Playable track. When two clips overlap each other on these tracks, the second clip overrides the first clip. It is possible to position a clip in such a way that it hides another clip.

Animation tracks, Animation Override tracks, and Audio tracks: When two clips overlap each other on these tracks, the first clip blends into the second clip. This is useful, for example, to create a seamless transition between two **Animation clips**.

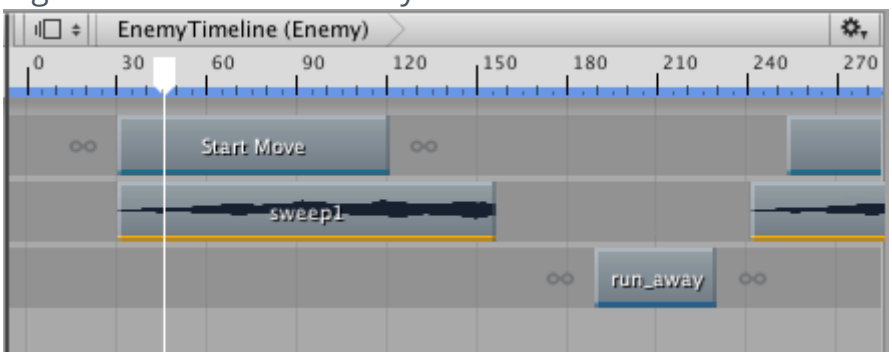
You can also position clips by inserting frames at the position of the Timeline Playhead. Right-click the Timeline Playhead, on the timeline above the Clips view, and choose **Insert** > **Frame** and a number of frames. This inserts frames in the Timeline asset at the position of the Timeline Playhead. Inserting frames only repositions the clips that start **after** the position of the Timeline Playhead.



Move the Timeline Playhead to where you want to insert frames



Right-click the Timeline Playhead and select ***Insert*** > ***Frame*** to move clips an exact number of frames



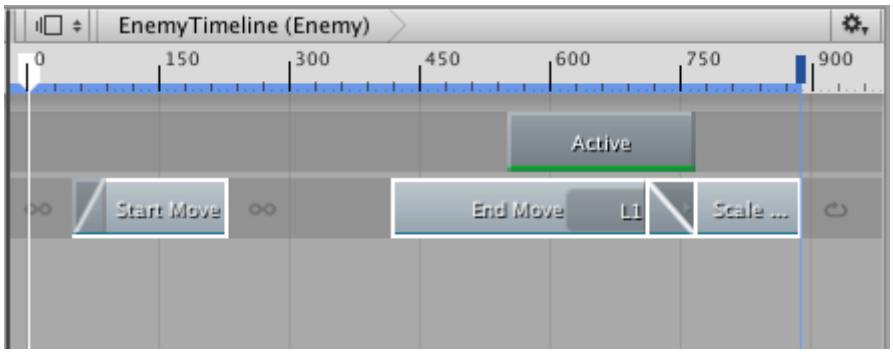
Only the clips that start after the Timeline Playhead are moved. In this example, inserting 100 frames at frame 45 affects the End Move, sweep2, and run_away clips.

2017–08–10 Page published with limited editorial review

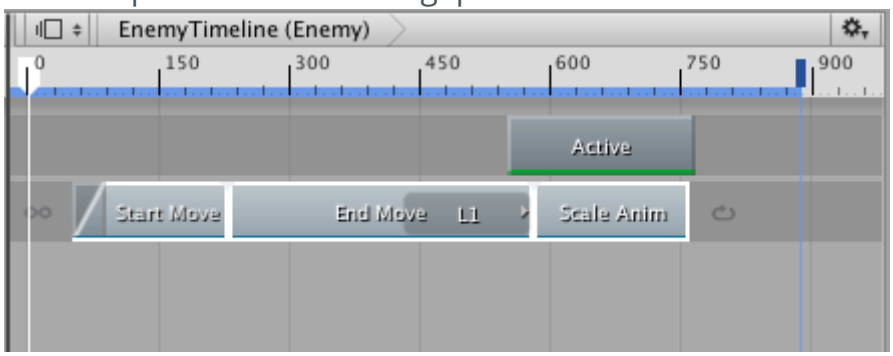
Tiling clips

Leave feedback

Tile clips to remove gaps, blends, and overlaps between clips on the same track. Tiling clips is useful if you want each clip to begin exactly where the previous clip ends. When tilling clips, at least two clips must be selected on the same track.



Three clips with blends and gaps are selected



Tiling removes blends and gaps between the selected clips

The selected clips are positioned based on the first selected clip. The first selected clip does not move. If you select multiple clips on multiple tracks, at least two clips must be selected on the same track for tile to have an affect.

2017–08–10 Page published with limited editorial review

Duplicating clips

Leave feedback

The **Timeline** Editor window supports the following different methods of duplicating clips:

Select a clip or multiple clips. Right-click in the **Clips view** and select **Duplicate** from the context menu.

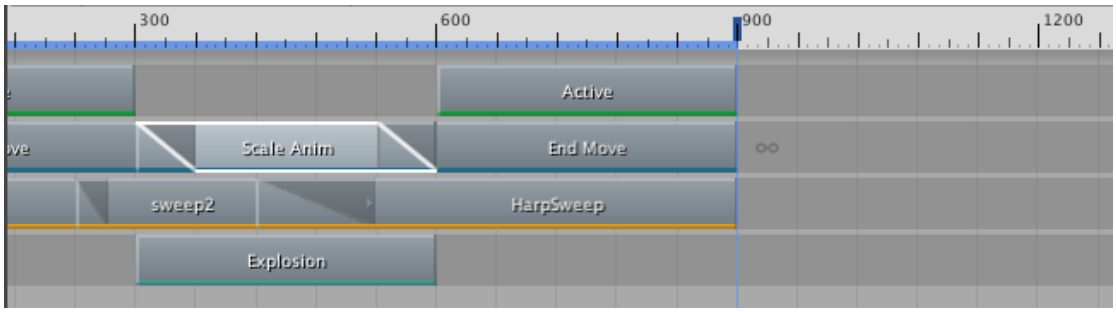
Select a clip or multiple clips. Hold Command/Control and press D.

Select a clip or multiple clips. Hold Command/Control and press C for copy, and press V for paste.

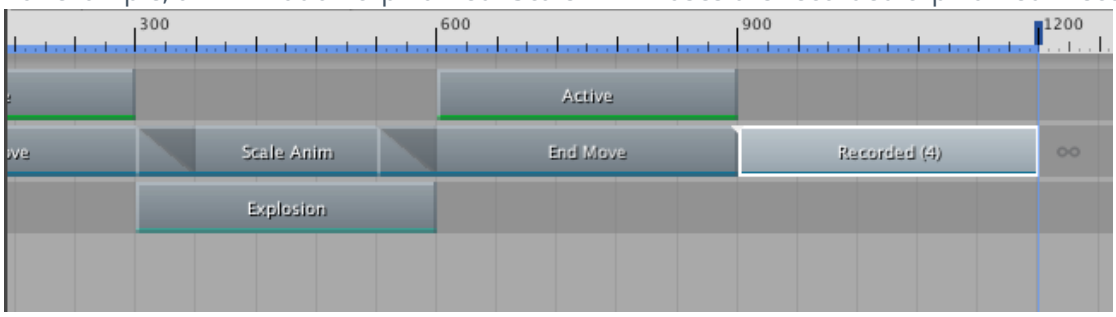
Right-click a clip, without selecting, and choose **Duplicate** from the context menu.

Duplicating clips copies each selected clip and places the duplicate after the last clip on its track. If you duplicate the clips used in a blend, the duplicated clips are tiled and the blend is removed.

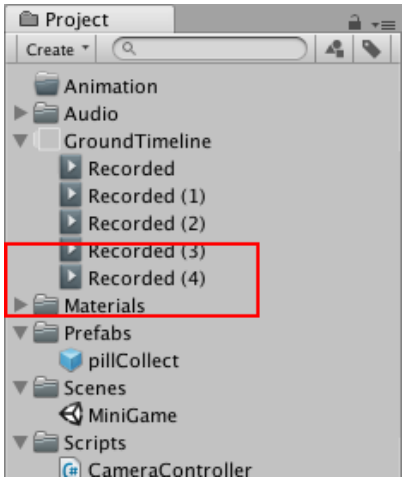
If you duplicate an **Animation clip** that uses a recorded clip as its source **asset**, a copy of the recorded clip is created and added to the **Timeline Asset**. The copy of the recorded clip only appears after you save the **scene** or project. The name of the copied recorded clip is based on the original recorded clip. The duplicated **Animation clip** is given the same name as the copied recorded clip.



For example, an Animation clip named “Scale Anim” uses the recorded clip named “Recorded (1)”.



Duplicating the “Scale Anim” clip places a copy of the “Recorded (1)” recorded clip at the end of the same track. The copy of the recorded clip is named “Recorded (4)” based on the number of recorded clips already associated with the Timeline Asset.



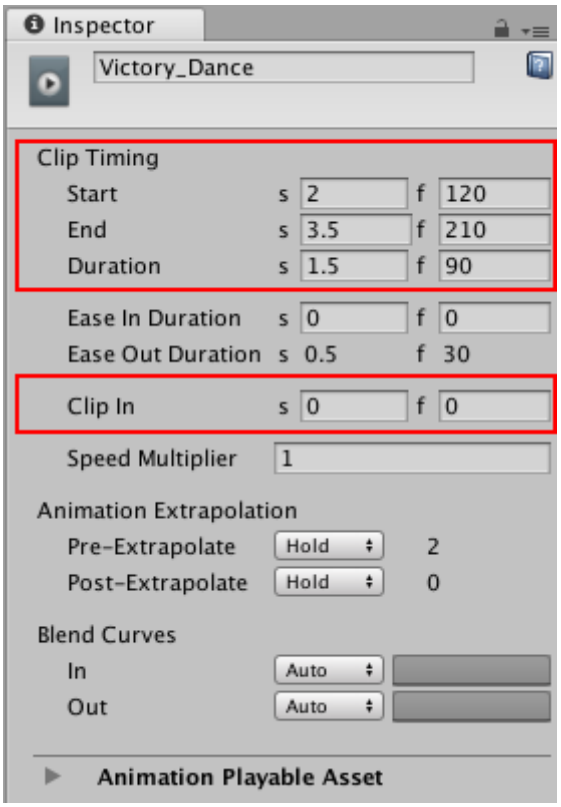
The new “Recorded (4)” recorded clip only appears in the Project window after you save the scene or project

2017–08–10 Page published with limited editorial review

Trimming clips

Leave feedback

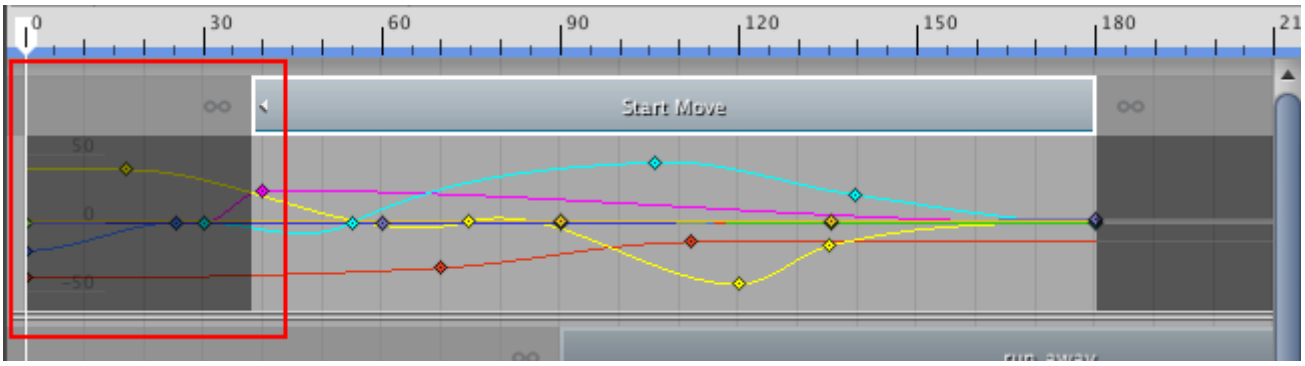
Drag the start or end of a clip to trim its duration. Dragging the start or end of a clip automatically selects the clip, showing its properties in the **Inspector** window. Use the Clip Timing properties in the Inspector window to set the start, end, duration, and offset (Clip In) of a clip to exact values.



Position and trim a clip by adjusting its Start, End, Duration, and Clip In properties in the Inspector window

Trimming the start of a clip

When trimming the start of an **Animation clip** or **Audio clip** in the **Clips view**, the start cannot be dragged before the start of the source **asset** that the clip is based on. Trimming an **Animation clip** or **Audio clip** after the start of the source **asset** selects the section of the source **asset** used by the clip.



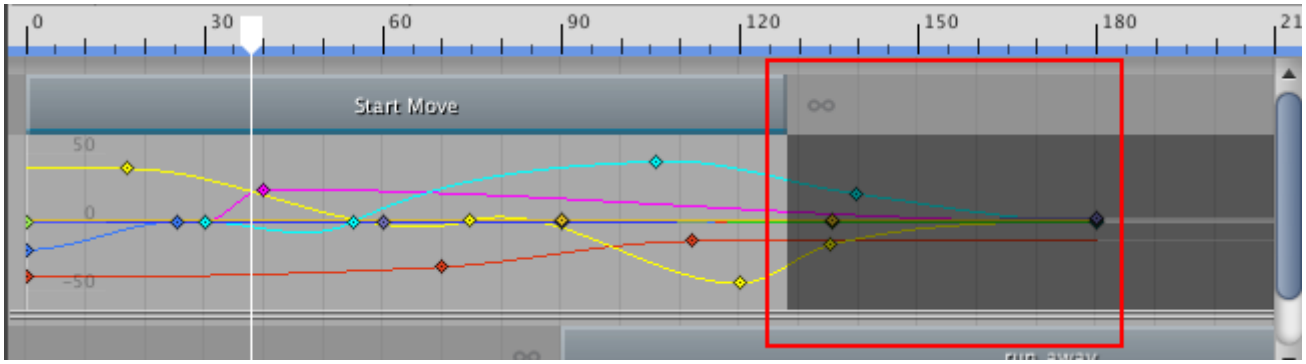
Trimming the start of an Animation clip selects the section of key animation used by the clip

Trimming a clip is non-destructive. Trim the clip again to modify its start to include the animation, or the audio waveform, cut off during a previous trim. You can also reset a clip to undo trims or other edits.

To trim the start of a clip to a precise time or frame, use the **Clip In** property in the Inspector window. Changing the **Clip In** property has the same effect as trimming the start of a clip after the start of its source asset.

Trimming the end of a clip

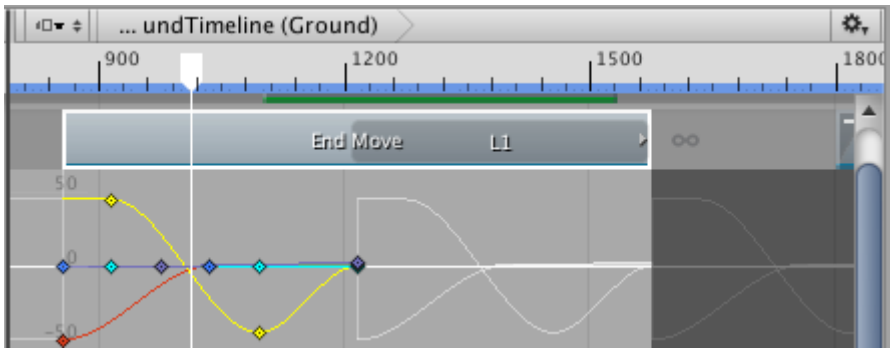
As with the start of the clip, trimming an Animation clip or Audio clip before the end of the source asset, trims the part of the source asset used by the clip.



Trimming the end of an Animation clip trims its key animation

If you trim the end of an Animation clip or Audio clip past the end of the source asset the clip is based on, the extra clip area either holds or loops, depending on the settings of the source asset.

For example, an Animation clip named “End Move” uses the motion file “Recorded(2)” as its source asset. The motion file “Recorded(2)” is set to loop. Trimming the end of the Animation clip past the end of the “Recorded(2)” source asset fills the extra clip area by looping “Recorded(2)”. A white **animation curve** shows the hold or loop.



A white animation curve indicates whether the extra clip area holds or loops data, depending on the source asset

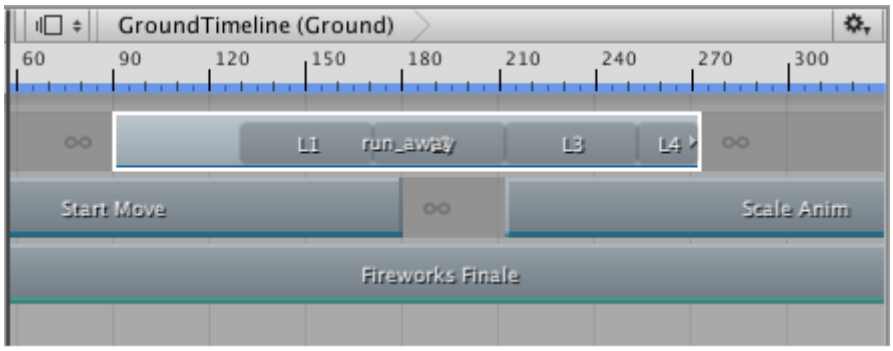
To choose whether the extra clip area holds or loops, select the source asset to change its settings in the Inspector window. Depending on the type of source asset, different properties control whether the source asset holds or loops.

If you are unsure which source asset is used by a clip, select the clip in the Clips view, Right-click and select **Find Source Asset** from the context menu. This highlights the source asset in the Project window.

Trimming the end of looping clips

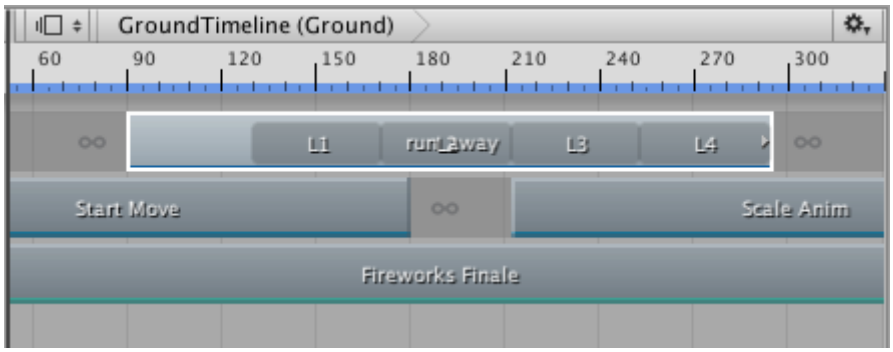
The **Timeline** Editor window provides special trimming options for Animation clips or Audio clips with loops. These special trim options either remove the last partial loop or complete the last partial loop.

For example, the Animation clip named run_away is over three times longer than the source asset on which it is based. Since the source asset is set to loop, the Animation clip loops the source asset until the Animation clip ends which results in a partial loop.

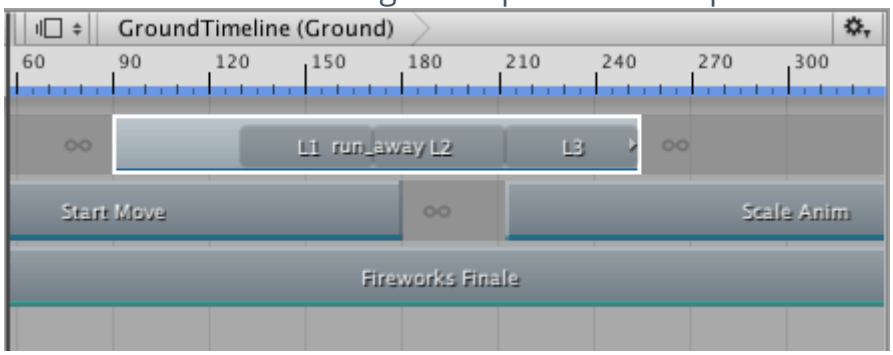


L1, L2, and L3 signify complete loops. The clip ends partially through the fourth loop, L4.

To extend the end of the clip and complete a partial loop, select the clip, Right-click and select **Editing** > **Complete Last Loop**. To trim the clip at the last complete loop, select the clip, Right-clip and select **Editing** > **Trim Last Loop**.



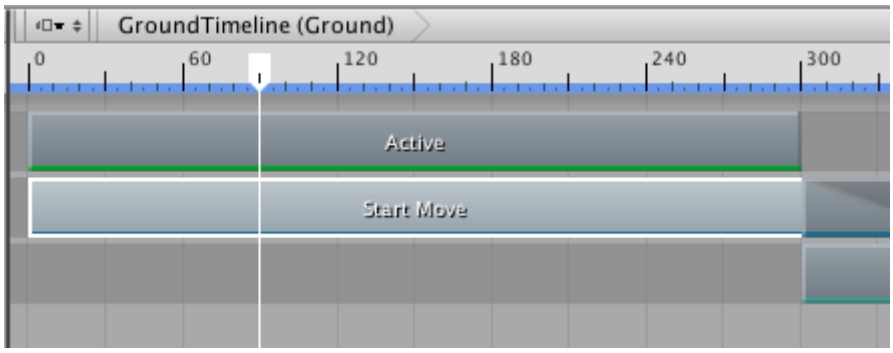
The result of select Editing > Complete Last Loop



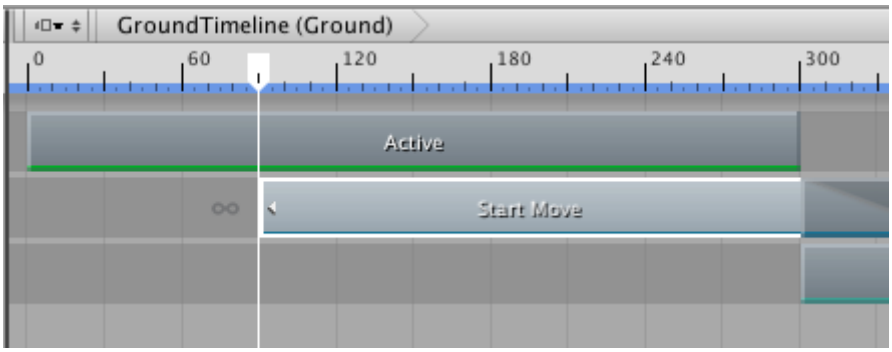
The result of select Editing > Trim Last Loop

Trimming with the Timeline Playhead

You can also trim a clip based on the location of the playhead. To trim using the playhead, position the playhead within the clip to be trimmed. Right-click the clip and select either **Editing** > **Trim Start** or **Editing** > **Trim End**. **Trim Start** trims the start of the clip to the playhead. **Trim End** trims the end of the clip to the playhead.



Move the Timeline Playhead within the clip



Right-click and select Editing > Trim Start to trim the start of the clip to the playhead

If you select clips on multiple tracks, only the selected clips that intersect the playhead are trimmed.

2017–08–10 Page published with limited editorial review

Splitting clips

Leave feedback

You can split a clip into two identical clips with different starts, ends, and durations. Splitting clips is non-destructive. The start or end of the clip can be extended to include the split animation or audio. You can also reset a clip to undo a split or other edits.

To split a clip, position the playhead within the clip to be split and either Right-click the clip and select **Editing** > **Split**, or press S. The clip is split into two separate clips that can be positioned, trimmed, and edited independently.

2017–08–10 Page published with limited editorial review

Resetting clips

Leave feedback

You can reset a clip to its original duration. To reset a clip, Right-click the clip and select **Editing** > **Reset Editing** from the context menu. Resetting a clip **does not reset** the following properties and settings:

Ease In Duration and Ease Out Duration

Clip In value

Clip speed

Animation **Extrapolation** settings

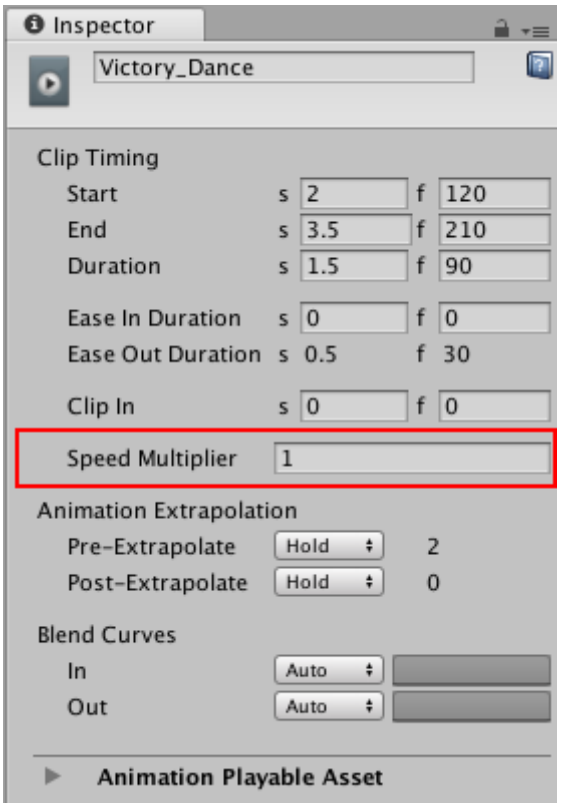
Blend Curves

2017–08–10 Page published with limited editorial review

Changing clip play speed

Leave feedback

Change the play speed of clip speed to accelerate or decelerate its audio, motion, animation, or particle effect. Changing the clip play speed affects the duration of the clip. You can only change the play speed for **Animation clips**, **Audio clips**, and Control clips. For all other clip types, the Speed options are disabled.



Speed Multiplier in the Inspector window

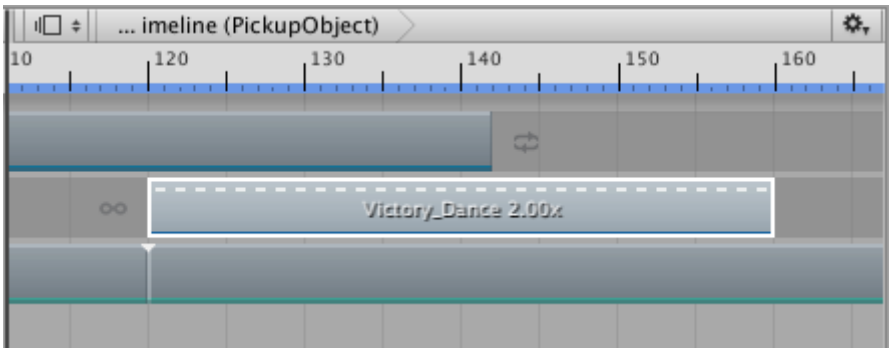
In the **Inspector** window, the **Speed Multiplier** property shows the play speed as a multiplier of the original clip speed. For example, changing the play speed of an 80 frame Animation clip to double speed changes the clip duration to 40 frames and sets the **Speed Multiplier** to 2.

To change the clip play speed, Right-click a clip and select one of the following options:

Speed > Double Speed to halve the clip duration. The clip plays at twice its current speed. A short-dashed line and a multiplication factor indicates an accelerated clip. Doubling the clip speed sets the **Speed Multiplier** property to double its current value.

Speed > Half Speed to double the clip duration. The clip plays at half its current speed. A long-dashed line and multiplication factor indicates a decelerated clip. Halving the clip speed sets the **Speed Multiplier** property to half its current value.

Speed > Reset Speed to reset the clip to its original duration. The clip plays at its original speed. Resetting the clip speed sets the **Speed Multiplier** property to 1.



A short-dashed line and multiplication factor of 2.00x indicates a clip playing at double its original speed

---

2017–08–10 Page published with limited editorial review

Setting gap extrapolation

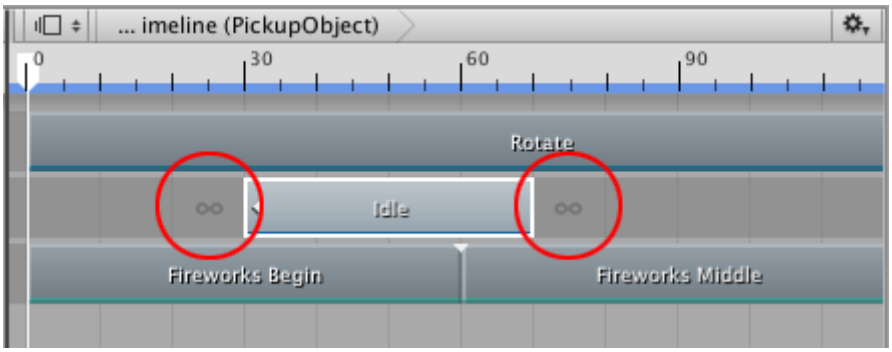
Leave feedback

Gap **extrapolation** refers to how an Animation track approximates animation data in the gaps before and after an **Animation clip**.

The main purpose for extrapolating animation data in the gaps between Animation clips is to avoid animation anomalies. Depending on the **GameObject** bound to the Animation track, these anomalies could be a **GameObject** jumping between two transformations, or a humanoid character jumping between different poses.

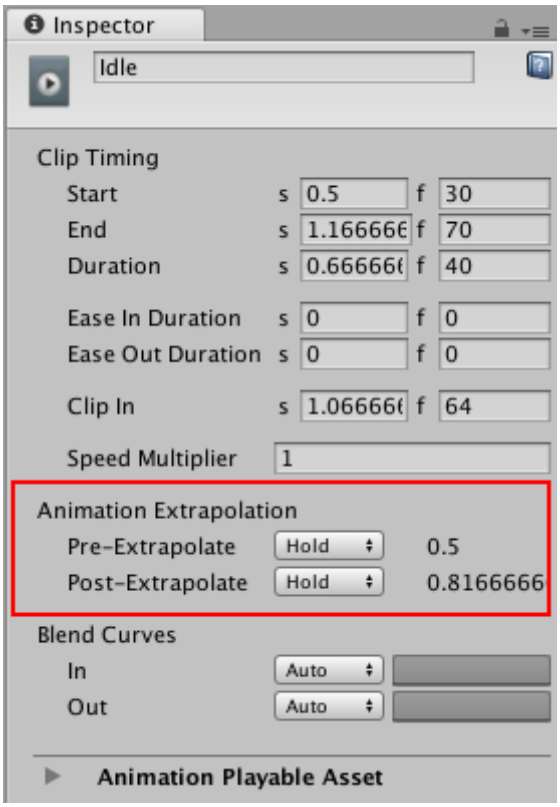
Each Animation clip has two **gap extrapolation** settings: **pre-extrapolate**, which controls how animation data is approximated in the gap before an Animation clip, and **post-extrapolate**, which controls how animation data extends in the gap after an Animation clip.

By default, both the pre-extrapolate and post-extrapolate settings are set to Hold. This sets the gap before the Animation clip to hold at the animation on the first frame, and the gap after the Animation clip to hold on the animation on the last frame. Icons before and after an Animation clip indicate the selected extrapolation modes.



Icons indicate the pre-extrapolate and post-extrapolate modes

To change the pre-extrapolate and post-extrapolate modes, select the Animation clip and use the Animation Extrapolation properties in the **Inspector** window.



Use Pre-Extrapolate and Post-Extrapolate to set the extrapolation modes for the selected Animation clip

If the selected Animation clip is the only clip on the Animation track, you can set the Pre-Extrapolate mode to one of the following options:

None: Turns off pre-extrapolation. In the gap before the selected Animation clip, the GameObject uses its transform, pose, or state from the **scene**. Selecting None is useful if, for example, you want to create an ease-in between the motion of a GameObject in the **scene** and an Animation clip. See Easing-in and Easing-out Clips for details.

Hold (default): In the gap before the selected Animation clip, the GameObject bound to the Animation track uses the values assigned at the start of the Animation clip.

Loop: In the gap before the selected Animation clip, the GameObject bound to the Animation track repeats the entire animation as a forward loop: from start to end. Use the **Clip In** property to offset the start of the loop.

Ping Pong: In the gap before the selected Animation clip, the GameObject bound to the Animation track repeats the entire animation forwards, then backwards. Use the **Clip In** property to offset the start of the loop. Changing the **Clip In** property affects the start of the loop, when looping forward, and the end of the loop, when looping backwards.

Continue: In the gap before the selected Animation clip, the GameObject bound to the Animation track either holds or loops the animation based on the settings of the source **asset**. For example, if the selected Animation clip uses the motion file “Recorded(2)” as its source asset and “Recorded(2)” is set to loop, then selecting Continue loops the animation according to the “Recorded(2)” Loop Time settings.

If the selected Animation clip is the only clip on the Animation track, you can set the Post-Extrapolate mode to one of the following options:

None: Turns off post-extrapolation. In the gap after the selected Animation clip, the GameObject uses its transform, pose, or state from the scene. Selecting None is useful if, for example, you want to create an ease-out between an Animation clip and the motion of a GameObject in the scene. See Easing-in and Easing-out Clips for details.

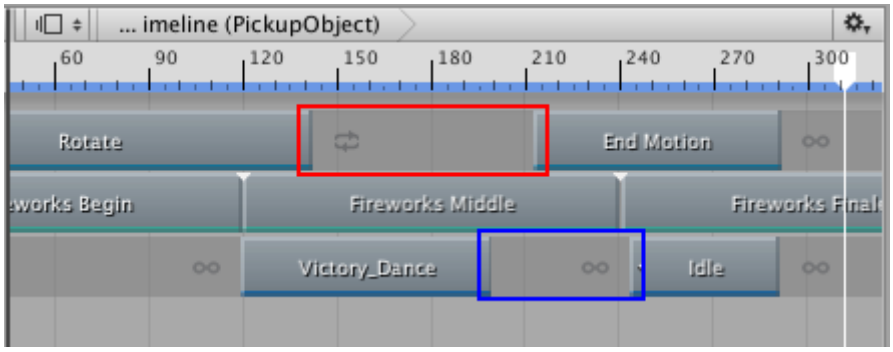
Hold (default): In the gap after the selected Animation clip, the GameObject bound to the Animation track uses the values assigned at the end of the Animation clip.

Loop: In the gap after the selected Animation clip, the GameObject bound to the Animation track repeats the entire animation as a forward loop: from start to end. Use the **Clip In** property to offset the start of the loop.

Ping Pong: In the gap after the selected Animation clip, the GameObject bound to the Animation track repeats the entire animation forwards, then backwards. Use the **Clip In** property to offset the start of the loop. Changing the **Clip In** property affects the start of the loop, when looping forward, and the end of the loop, when looping backwards.

Continue: In the gap after the selected Animation clip, the GameObject bound to the Animation track either holds or loops the animation based on the settings of the source asset. For example, if the selected Animation clip uses the motion file “Recorded(2)” as its source asset and “Recorded(2)” is set to loop, then selecting Continue loops the animation according to the “Recorded(2)” Loop Time settings.

When an Animation track contains a gap between two Animation clips, the **Post-Extrapolate** setting of the left clip sets the gap extrapolation. If the **Post-Extrapolate** setting of the clip to the left of a gap is set to None, the **Pre-Extrapolate** setting of the right clip sets the gap extrapolation. Icons before and after Animation clips indicate whether the extrapolation for a gap is taken from the **Post-Extrapolate** setting of the clip to the left or from the **Pre-Extrapolate** setting of the clip to the right.



First track (red): gap extrapolation from Post-Extrapolate of the left clip. Third track (blue): gap extrapolation from Pre-Extrapolate of the right clip.

2017–08–10 Page published with limited editorial review

Easing-in and Easing-out clips

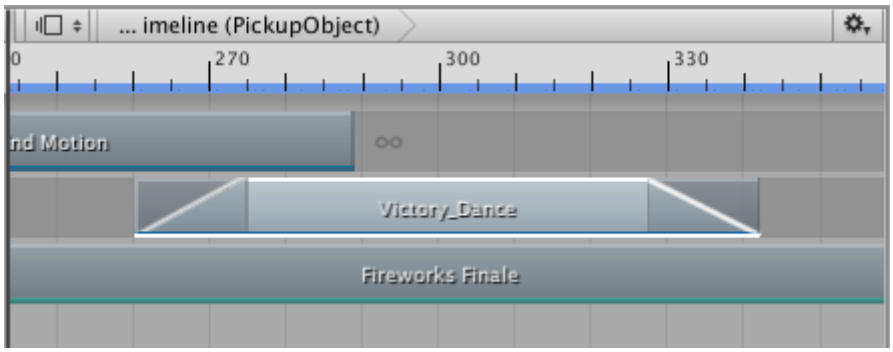
Leave feedback

Ease-in and ease-out a clip to create a smooth transition between a clip and its surrounding gaps. To create an ease-in or ease-out transition, use one of the following methods:

Hold Control/Command and drag the start of a clip to the right to add an ease-in.

Hold Control/Command and drag the end of a clip to the left to add an ease-out.

Select the clip and, in the **Inspector** window, set either the **Ease In Duration** or the **Ease Out Duration**.



Ease-in and ease-out an Animation clip to transition between its animation and its gaps. All ease-in and ease-out transitions are represented by a linear curve.

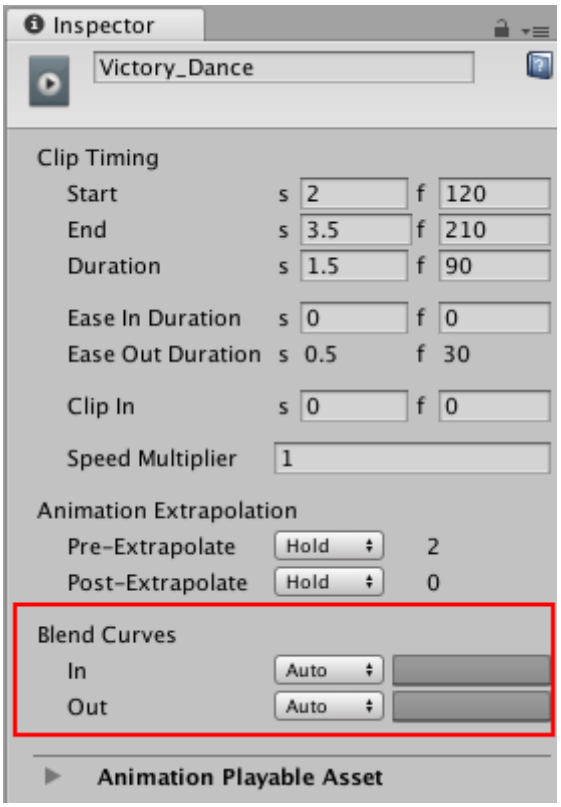
What an ease-in or ease-out transition effects differs depending on the track:

On an Animation track or an Animation Override track, ease-in an **Animation clip** to create a smooth transition between the animation in the gap before the clip and the **Animation clip**. Ease-out an **Animation clip** to create a smooth transition between the **Animation clip** and the animation in the gap after the clip. There are many factors that determine what animation occurs in the gap before and after an **Animation clip**. See Setting gap extrapolation for details.

On an Audio track, ease-in an **Audio clip** to fade in the volume of the audio waveform. Ease-out an **Audio clip** to fade out the volume of the audio waveform specified by the **Audio clip**.

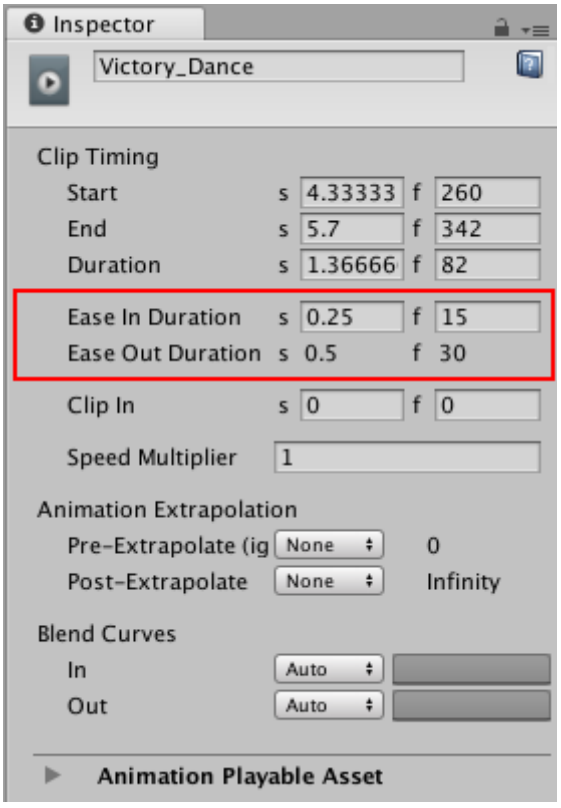
On a Playable track, ease-In a Playable clip to fade in the effect or script in the Playable clip. Ease-Out a Playable clip to fade out the effect or script in the Playable clip.

Although the **Clips view** represents an ease-in or ease-out as a single linear curve, every ease-in or ease-out transition is actually set to a gradually easing-in or easing-out curve by default. To change the shape of either the ease-in curve (labelled In) or the ease-out (labelled Out) curve, use the Blend Curves in the Inspector window.



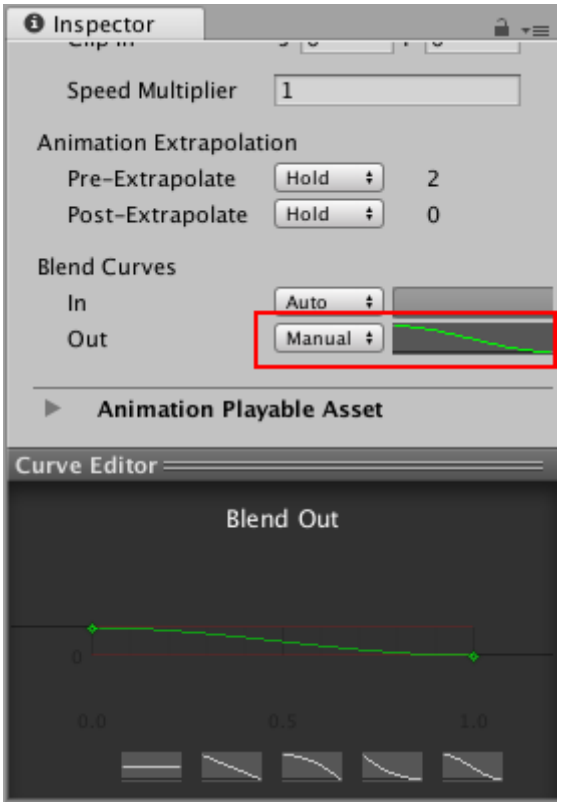
Use the Blend Curves to customize ease-in or ease-out transitions

Note that the **Blend Curves** might affect the **blend area** used for blending between two clips. The **Ease In Duration** and **Ease Out Duration** properties indicate whether the **Blend Curves** affect an ease-in, an ease-out, or a blend. For example, If the **Ease Out Duration** is editable, then the **Blend Out curve** (labelled **Out**) affects the curve used by an ease-out transition. If the **Ease Out Duration** cannot be edited, then the Blend Out curve (labelled **Out**) affects the **out-going clip** in a blend between two clips.



Ease Out Duration cannot be edited, therefore the Out curve affects the blend area between two clips

To customize either the ease-in or ease-out transition, use the drop-down menu to switch from **Auto** to **Manual**. With **Manual** selected, the Inspector window shows a preview of the blend curve. Click the preview to open the Curve Editor below the Inspector window.



Select Manual and click the preview to open the Curve Editor

The Curve Editor is the same editor that is used to customize the shape of the blend curves when blending between clips.

When creating an ease-in or an ease-out with Animation clips, the Animation clip blends between its gaps and the Animation clip. The following factors affect the values of animated properties in the gaps surrounding an Animation clip:

The pre-extrapolate and post-extrapolate settings for the Animation clip and for other Animation clips on the same track.

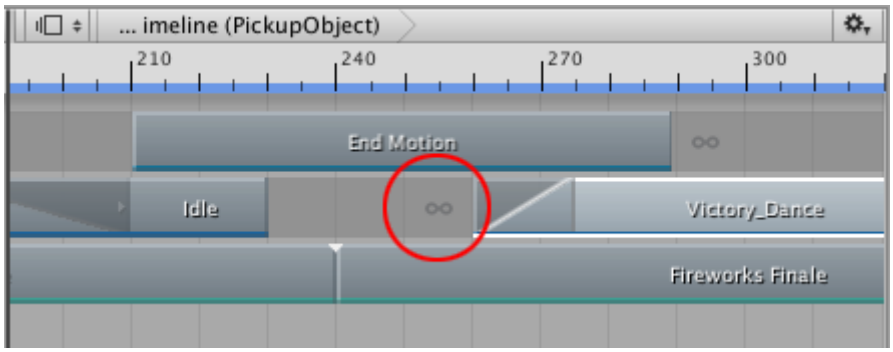
Animation clips on other Animation tracks that are bound to the same **GameObject**.

The position or animation of the GameObject in the **scene**, outside the **Timeline Asset**.

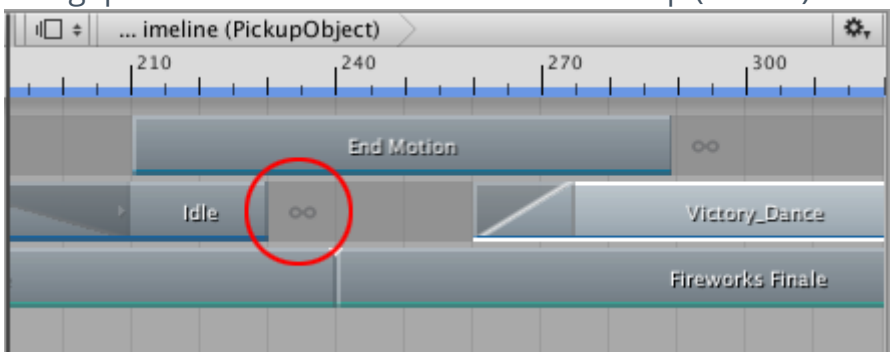
Gap extrapolation and easing clips

To successfully ease-in or ease-out an Animation clip, the **gap extrapolation** must not be set based on the Animation clip being eased-in or eased-out. The **gap extrapolation** must either be set to None or it must be set by other Animation clip.

For example, the following ease-in has no effect because the Pre-Extrapolate for the Victory_Dance clip is set to Hold. This means that the ease-in creates a transition between the first frame of the Animation clip and the rest of the Animation clip.



The gap is set to Hold from the Animation clip (circled). The ease-in has no effect.

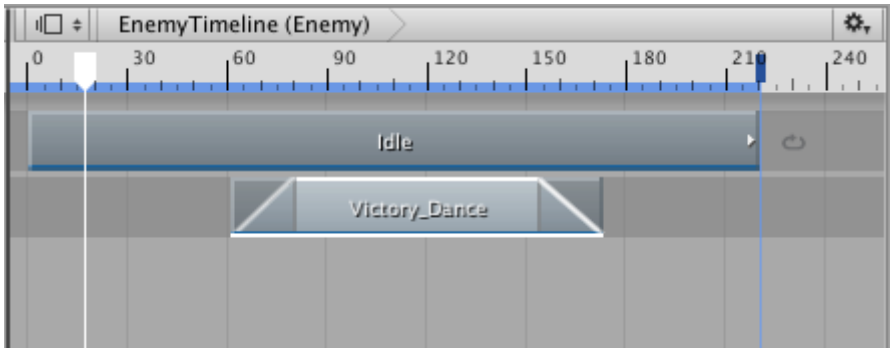


To ease-in from the Idle clip, set pre-extrapolate for the Victory_Dance clip to None. The ease-in gap uses the post-extrapolate mode from the Idle clip (circled).

Overriding Animation tracks with ease-in and ease-out

If the gap **extrapolation** is set to None, and there is a previous track bound to the same GameObject, the animation in the gap is taken from the previous track. This is useful for creating a smooth transition between two Animation clips on different tracks.

For example, if two Animation tracks are bound to the same GameObject and a clip on the second track contains an ease-in, the ease-in creates a smooth transition between the animation on the previous track and the animation on the second track. To successfully override animation on a previous track, the gap extrapolation for the second track must be set to None.



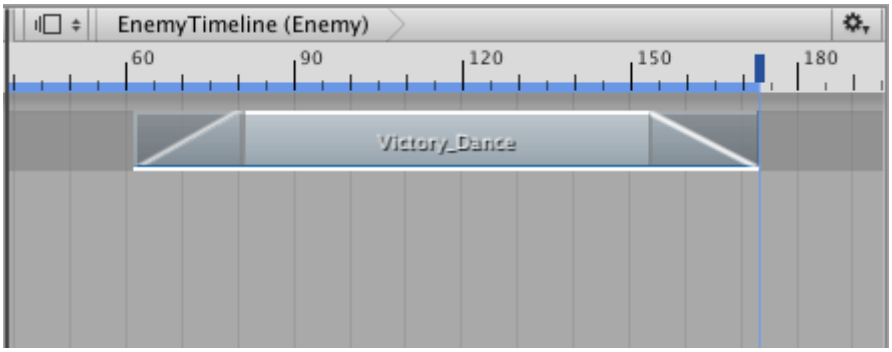
The Animation clip on the first track is a repeated idle cycle where the humanoid GameObject stands still. The Animation clip in the second track eases-in the Victory_Dance motion and ease-out to return back to the idle cycle.

Overriding the scene with ease-in and ease-out

In the scene, if a GameObject is controlled with an **Animator Controller**, you can use an ease-in or ease-out transition between the Animation clip and the **Animator Controller**.

For example, if a **Timeline Asset** contains a single track with a single Animation clip and all its gap extrapolation settings are set to None, the gap uses the position or animation of the GameObject from the scene.

This position or animation is the GameObject as set in the scene. If the GameObject uses an Animator Controller to control its animation state, the gap is set to the current animation state. For example, if the GameObject is a character walking in the scene, a Timeline Asset could be setup to ease-in animation to override the walking animation state. The ease-out returns the GameObject to the animation state according to the Animator Controller.



A single Animation clip with all gap extrapolation set to None eases-in and ease-out the GameObject between its position or animation in the scene and the Animation clip

---

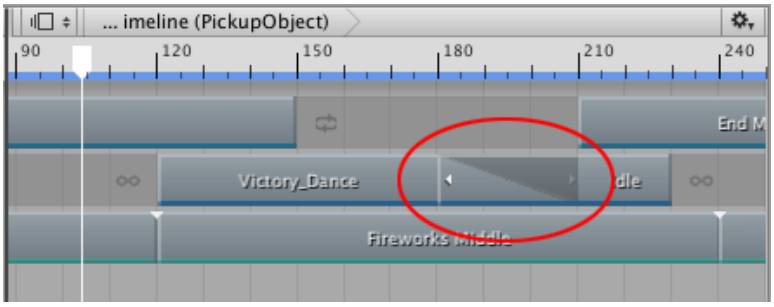
2017–08–10 Page published with limited editorial review

Blending clips

Leave feedback

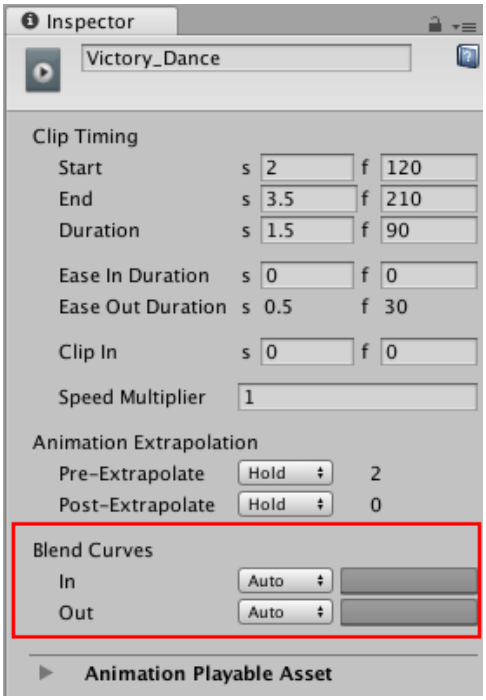
Blend two clips on the same track to create a smooth transition between two **Animation clips**, two **Audio clips**, or two Playable clips. To blend two clips, position or trim one clip until it overlaps the other clip.

In a blend, the first clip is referred to as the **out-going clip** and the second clip is referred to as the **incoming clip**. The area where the out-going clip transitions to the incoming clip is referred to as the **blend area**. The blend area sets the duration of the transition.



The blend area shows the transition between the out-going clip and incoming clip

Although the **Clips view** represents a blend area as a single linear curve, the transition between clips is actually comprised of two blend curves. The blend curve for the out-going clip is referred to as the **Blend Out** curve. The blend curve for the incoming clip is referred to as the **Blend In** curve. By default, each blend curve is automatically set to an ease-in and ease-out curve.



Use Blend Curves to customize the blend area

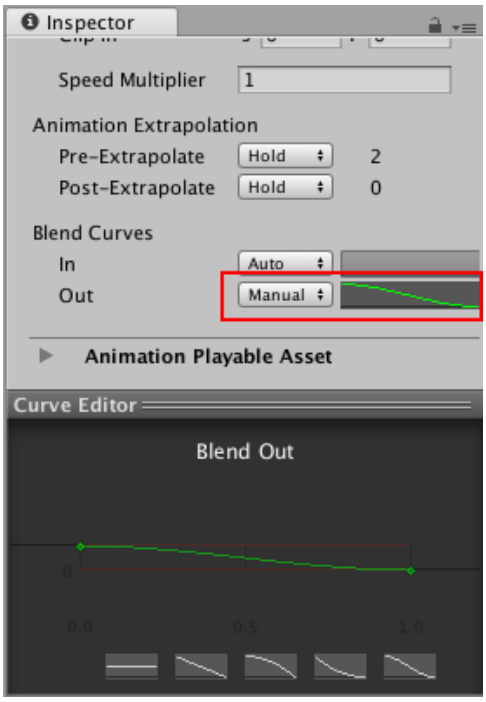
Use the **Blend Curves** in the **Inspector** window to change the shape for either the Blend In or **Blend Out curve** of the selected clip. However, the **Inspector** window only allows for editing the properties of one clip at a time. It is not possible to simultaneously customize both blend curves from the same blend area.

To customize the **Blend Curves** for the transition between two clips, do the following:

Select the out-going clip to customize its Blend Out curve (labelled **Out**).

Select the in-coming clip to customize its **Blend In curve** (labelled **In**).

To customize either the Blend Out curve or Blend In curve, use the drop-down menu to switch from **Auto** to **Manual**. With **Manual** selected, the Inspector window shows a preview of the blend curve. Click the preview to open the Curve Editor below the Inspector window.



Select Manual and click the preview to open the Curve Editor

Use the Curve Editor to customize the shape of the blend curve. By default, the blend curve includes a key at the beginning of the curve and a key at the end of the curve. The Curve Editor provides the following different methods of modifying the blend curve:

Select the key at the start of the blend curve and use the tangent handles to adjust the **interpolation** between keys.

Select the key at the end of the blend curve and use the tangent handles to adjust the interpolation between keys.

Add additional keys to change the shape of the blend curve by adding more interpolation points. Adding keys in the Curve Editor is the same as adding keys in the Curves view.

Right-click a key to delete or edit the key. Editing keys in the Curve Editor is the same as editing keys in the Curves view. Note that the first and last keys cannot be deleted.

Select a shape template from the bottom of the Curve Editor.

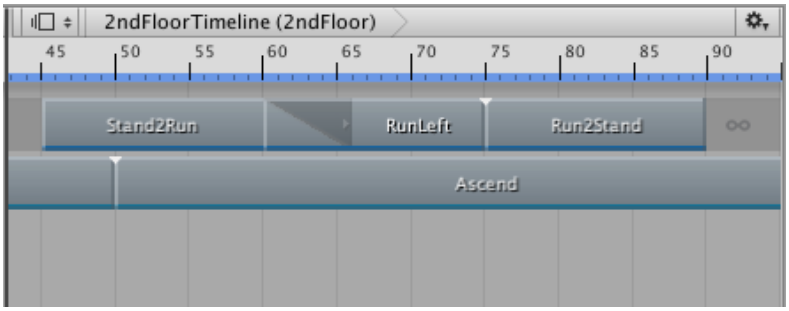
The Curve Editor also includes shape templates based on whether you are modifying the Blend In curve or the Blend Out curve. Select a shape template to change the blend curve to the selected shape template.

2017–08–10 Page published with limited editorial review

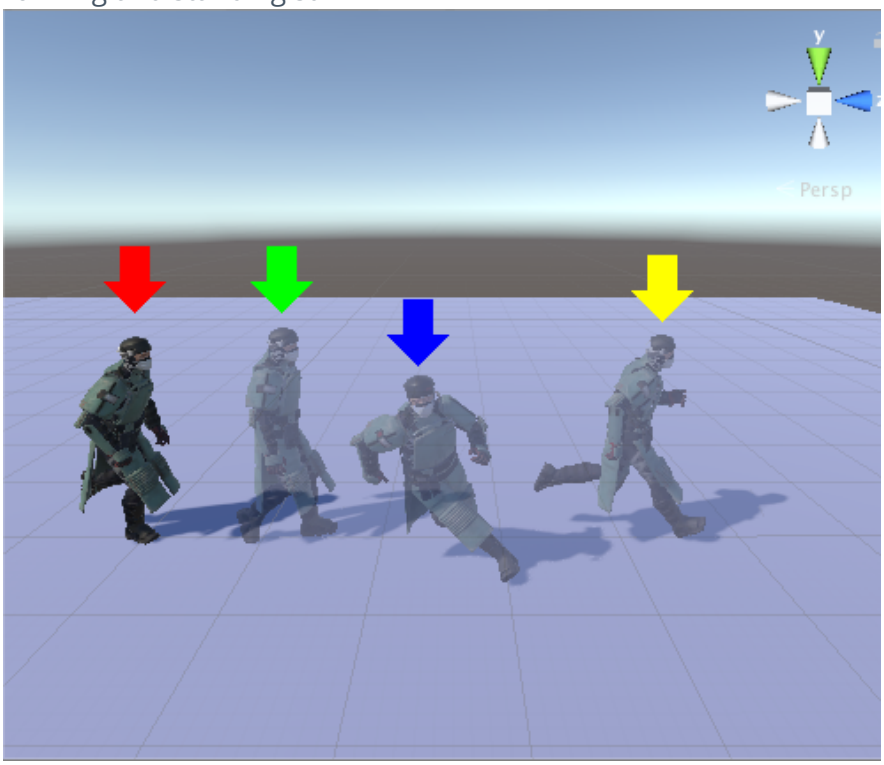
Matching clip offsets

Leave feedback

Every **Animation clip** contains key animation, or motion, that animates the **GameObject**, or humanoid, bound to the Animation track. When you add an **Animation clip** to an Animation track, its key animation or motion does not automatically match the previous clip or the next clip on the Animation track. By default, each **Animation clip** begins at the position and rotation of the **GameObject**, or humanoid, at the beginning of the **Timeline instance**.



For example, three Animation clips create an animation sequence that starts with a clip of a humanoid standing and running, continues with a clip of the humanoid running while turning left, and ends with the humanoid running and standing still

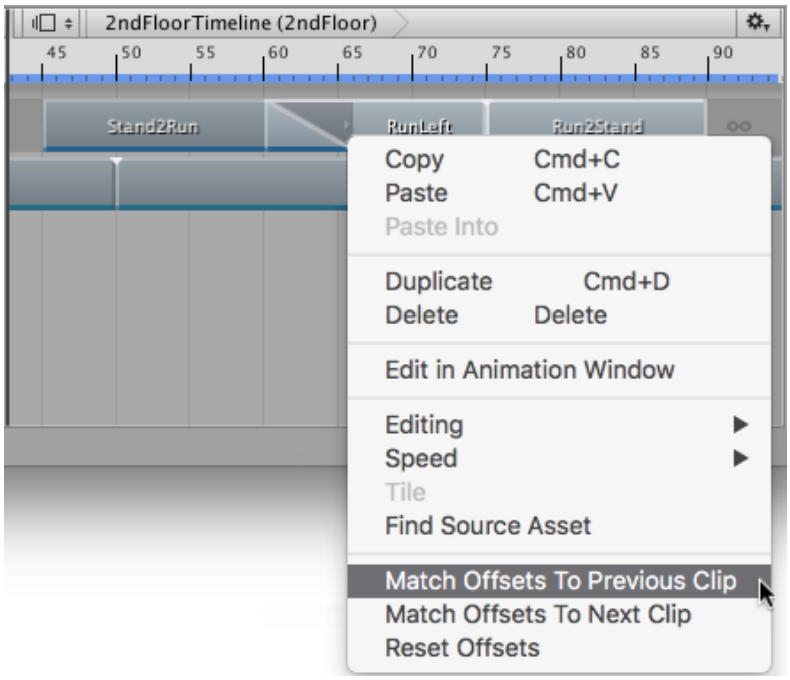


Each Animation clip begins at the position and rotation of the humanoid at the start of the Timeline instance (red arrow). The three Animation clips, Stand2Run, RunLeft, and Run2Stand, all begin at the red arrow but end at the green, blue, and yellow arrows, respectively.

For an animation sequence to flow seamlessly between adjacent Animation clips, you must match each Animation clip with its previous or next clip. Matching clips adds a position and rotation offset for each Animation clip, referred to as the Clip **Root Motion** Offset. The following sections describe how to match two Animation clips or many Animation clips.

Matching two clips

To match the root motion between two clips, RIght-click the Animation clip that you want to match. From the context menu, select either Match Offsets to Previous Clip or Match Offsets to Next Clip.



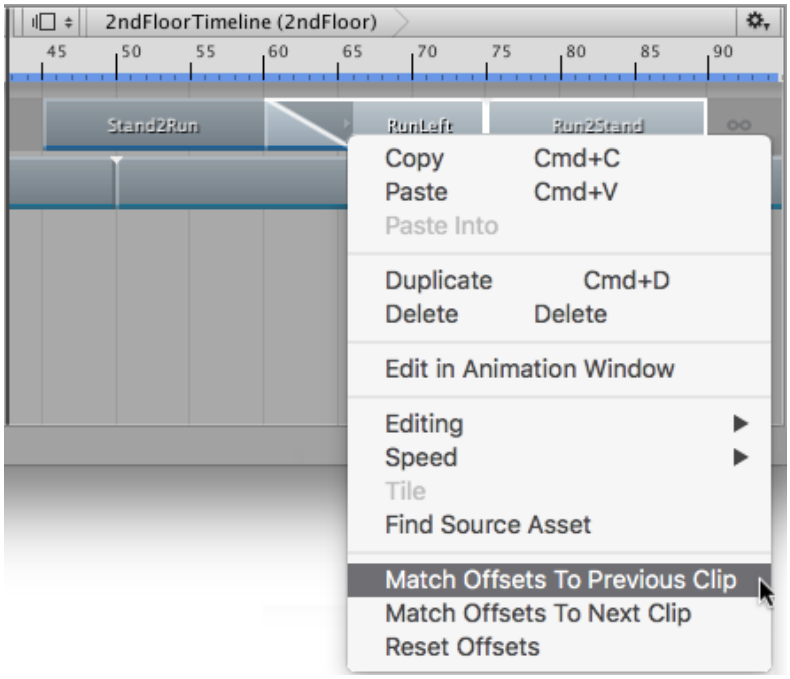
For example, Right-click the middle Animation clip, named “RunLeft”, to match its offsets to either the previous clip or the next clip

The context menu only displays the Match Offset options available for the clicked Animation clip. For example, if there is a gap **before** the clicked Animation clip, only the Match Offsets to Next Clip menu item is available.

When you are matching offsets for a single Animation clip, it is not necessary to select the Animation clip first, but you must Right-click the Animation clip that you want to match. For example, if you Right-click on an Animation clip that is not selected, the clicked clip is matched and any selected Animation clips are ignored.

Matching many clips

To match the root motion of many clips, select the Animation clips that you want to match and RIght-click on one of the selected clips. From the context menu, select either Match Offsets to Previous Clip or Match Offsets to Next Clip.



For example, select the “RunLeft” and “Run2Stand” clips. Right-click one of the selected clips, and select Match Offsets to Previous Clips, to match the RunLeft clip with the previous clip Stand2Run, and to match Run2Stand with the previous clip RunLeft.

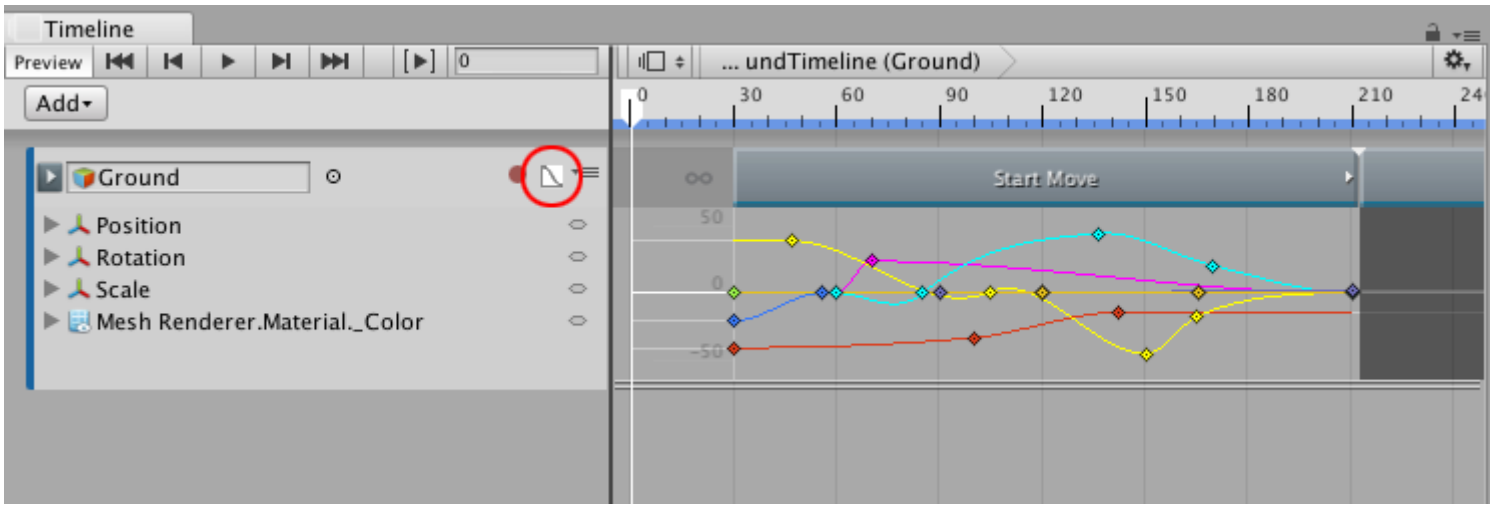
2017–12–07 Page published with no editorial review

Curves view

Leave feedback

The **Curves view** shows the **animation curves** for **Infinite clips** or for **Animation clips** that have been converted from **Infinite clips**. Use the **Curves view** for basic animation editing such as modifying existing keys, adding new keys, adjusting tangents, and changing the **interpolation** between keys.

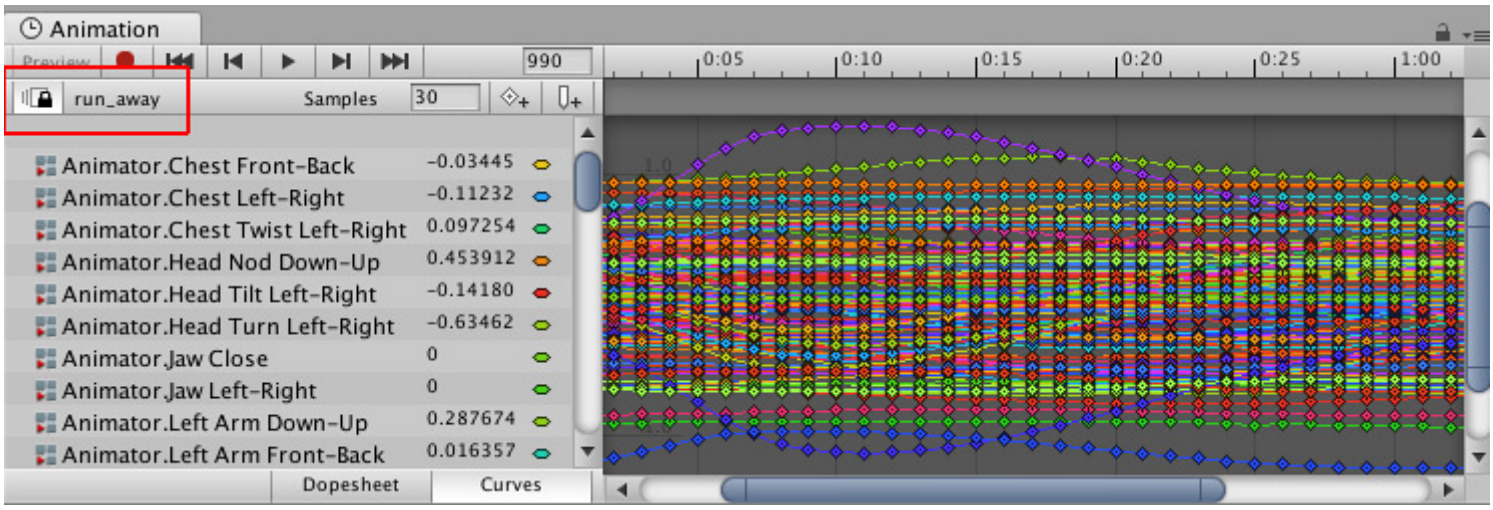
To view animation curves for an Infinite clip, click the Curves icon next to the Track name. To view animation curves for an Animation clip, select the Animation clip and click the Curves icon. The Curves view is similar to Curves mode in the Animation window.



The Curves icon (circled) shows and hides the Curves view

The Curves icon does not appear for Animation tracks with **humanoid animation** or imported animation. To view and edit key animation for humanoid or imported Animation clips, Right-click an Animation clip and select **Edit in Animation Window** from the context menu. You can also Double-click the Animation clip. The Animation window appears, linked to the **Timeline** Editor window.

When in linked mode, the Animation window shows a Linked icon and the name of the Animation clip being edited. Click the Linked icon to stop editing the Animation clip and to release the Animation window from linked mode.



Animation window linked to the Timeline Editor window, indicated by the Linked icon and Animation clip name

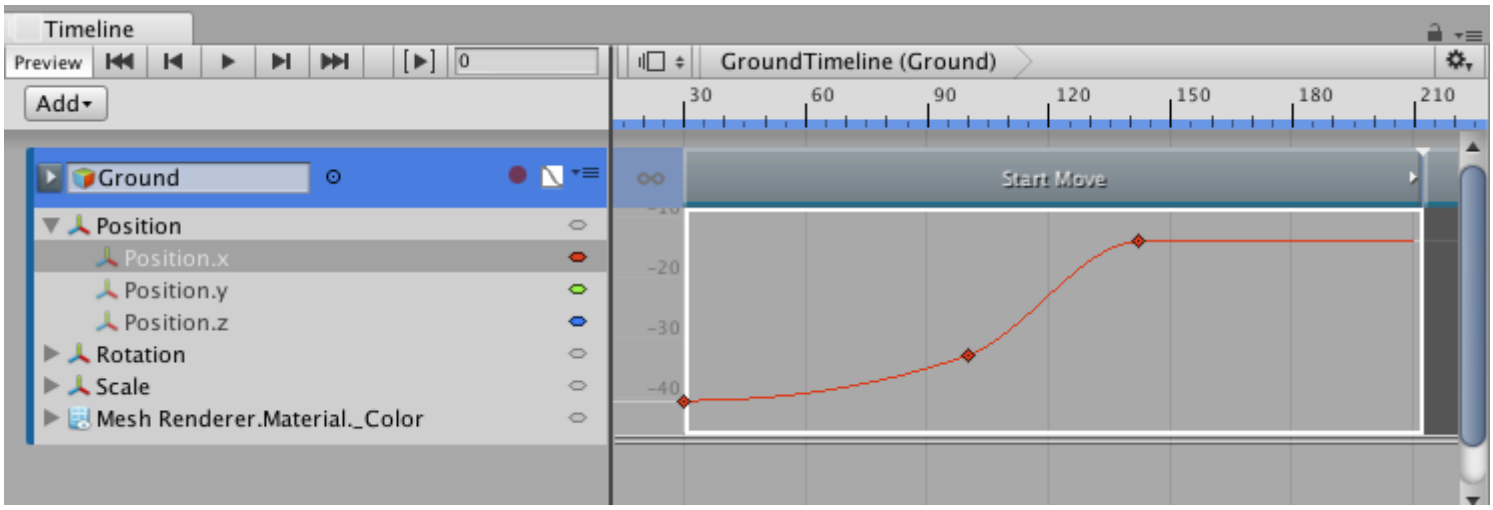
2017–08–10 Page published with limited editorial review

Hiding and showing curves

Leave feedback

For the selected **Animation clip**, the **Curves view** includes a hierarchical list of the properties with **animation curves**. Expand, select, and deselect the properties in this list to filter which **animation curves** are shown in the **Curves view**.

For example, to show only the animation curves for position along the X-axis, expand **Position** and select the **Position.x** property. Press F to frame the animation curve for the **Position.x** property.



Curves view showing the animation curve for the Position.x property

The Curves view supports the following methods of selecting and deselecting animation curves:

Click the Triangle icon of a parent property to expand and collapse its list of child properties.

Hold Shift and click to select contiguous properties.

Hold Command/Control and click to select discontiguous properties. Hold Command/Control and click a selected property to deselect it.

2017–08–10 Page published with limited editorial review

Navigating the Curves view

Leave feedback

Use one of the following methods to pan, zoom, resize, or frame the **animation curves** and keys in the **Curves view**:

To pan, Middle-drag, or hold Alt and drag.

To zoom vertically, move the scroll-wheel, or hold Alt and Right-drag.

To zoom horizontally, hold Command/Control and zoom vertically.

To resize the Curves view, drag the double line separating the Curves view from the next track in the **Track list**.

To frame only selected animation curves or selected keys, press F.

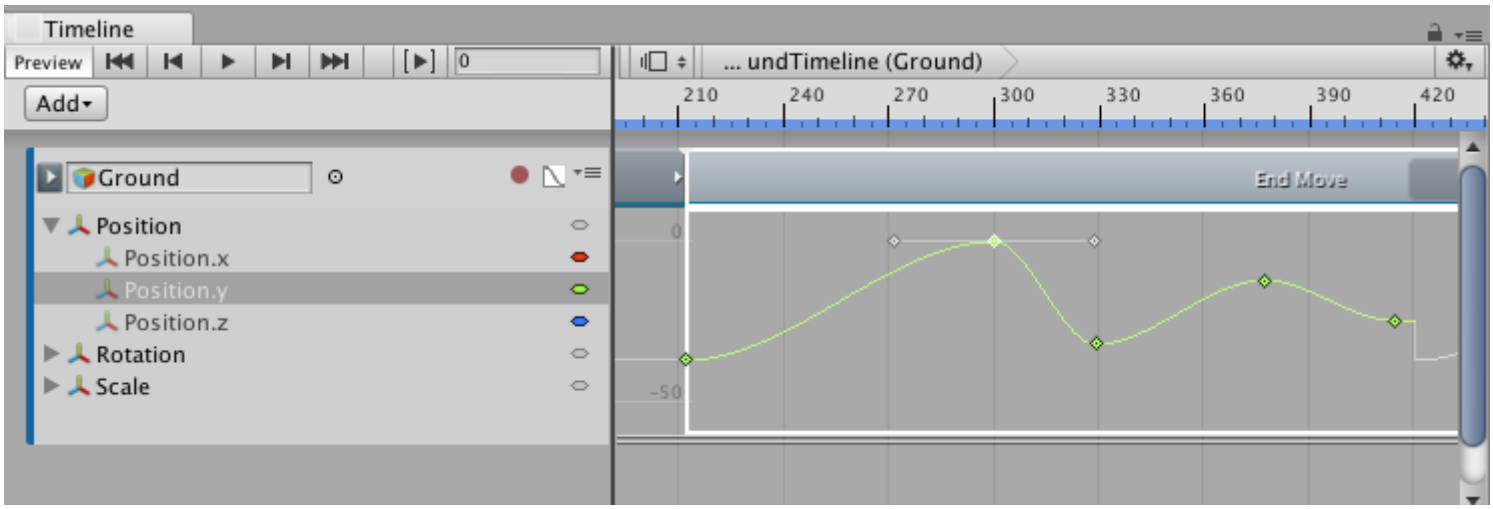
To frame all animation curves or keys, press A.

2017–08–10 Page published with limited editorial review

Selecting keys

Leave feedback

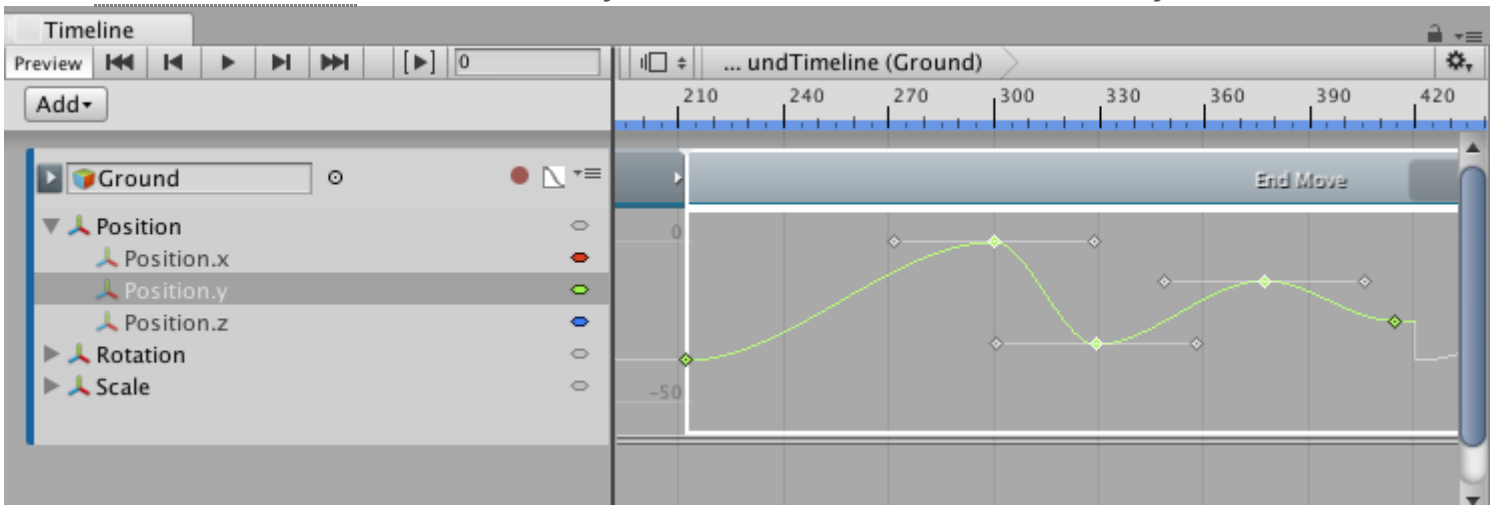
Click to select a single key. Selecting a key deselects all other selected keys. The **Curves view** displays the selected key with its tangents.



Click to select a single key. A selected key shows its tangents.

The Curves view provides the following methods for selecting keys:

Hold Shift and click to select contiguous keys. For example, to select contiguous keys along the same **animation curve**, click the first key, then hold Shift and click the last key.



Hold Shift and click a key to select contiguous keys

Hold Command/Control and click to select discontiguous keys. Hold Command/Control and click a selected key to deselect it.

Click and drag on an empty spot in the Curves view to draw a selection rectangle. This selects all keys within the rectangle. Hold down Shift while drawing the selection rectangle to add keys to the current selection.

Double-click a selected key to select all keys on the same animation curve.

2017–08–10 Page published with limited editorial review

Adding keys

Leave feedback

The **Curves view** provides the following methods for adding keys:

Right-click on an **animation curve** and select **Add Key**. This method adds a key at the location of the Right-click.

Double-click on an animation curve. This method adds a key at the location of the Double-click.

2017–08–10 Page published with limited editorial review

Editing keys

Leave feedback

Edit a key to change its time, value, or both. The **Curves view** provides the following different methods for editing a key:

Right-click a key and select **Edit** from the context menu to enter specific values for time and value.

Select a key and press Enter to enter specific values.

Select and drag a key to change it time and value.

Drag a key vertically, then press Shift to snap the key on the vertical axis. This changes the value of the key, but not its time.

Drag a key horizontally, then press Shift to snap the key on the horizontal axis. This changes the time of the key, but not its value.

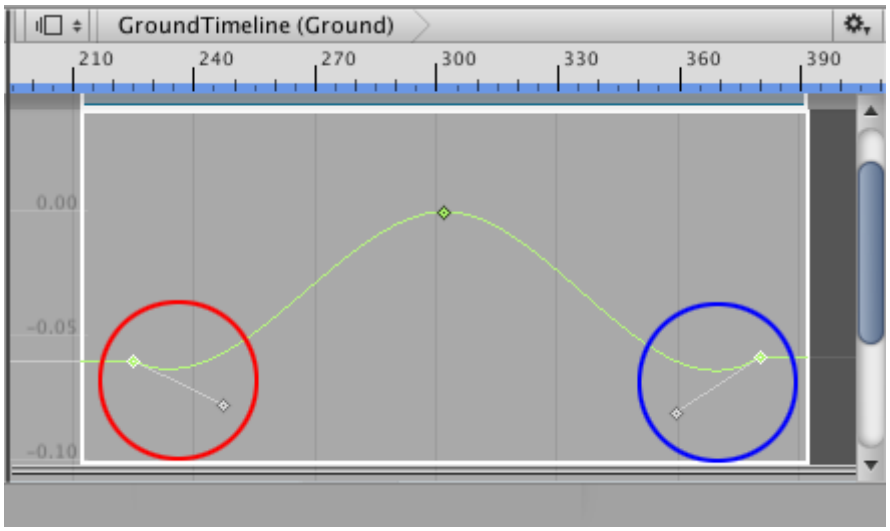
2017–08–10 Page published with limited editorial review

Changing interpolation and shape

Leave feedback

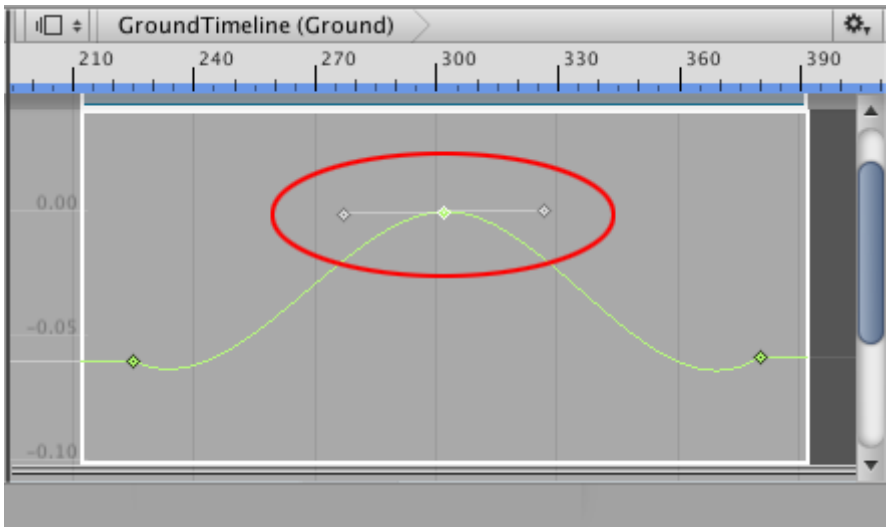
Every key has one or two tangents that control the interpolation of the **animation curve**. The term **interpolation** refers to the estimation of values that determine the shape of the **animation curve** between two keys.

Whether a key has one of two tangents depends on the location of the key on the animation curve. The first key has only a right tangent that controls the interpolation of the animation curve after the key. The last key has only a left tangent that controls the interpolation of the animation curve before the last key.



The first key (red) has only a right tangent, and the last key (blue) has only a left tangent

All other keys have two tangents where the left tangent controls the interpolation before the key, and the right tangent controls the interpolation after the key. By default, tangents are joined. Dragging one tangent affects the position of both tangents, and the interpolation of the animation curve both before and after the key.

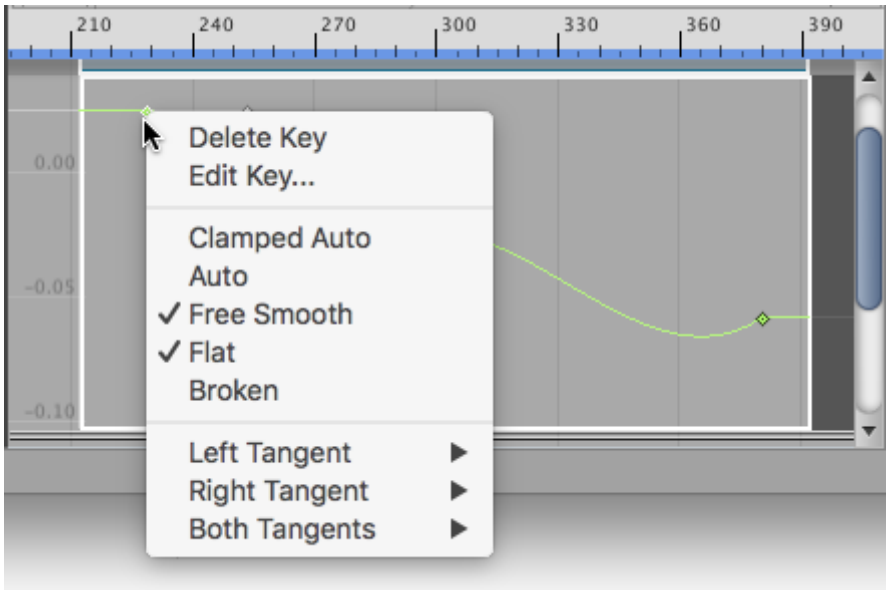


Keys that are neither the first key nor last key have joined tangents by default. Dragging either tangent changes the interpolation of the animation curve both before and after the key.

Dragging a tangent may also change the interpolation mode of the animation curve. For example, most keys are set to the Clamped Auto interpolation mode which automatically smooths animation curve as it passes through the key. If you drag a tangent of a key set to Clamped Auto, the interpolation mode changes to Free Smooth.

The term **interpolation mode** refers to the interpolation algorithm that determines which shape to use when drawing the animation curve.

To view the interpolation mode for a key, select the key and Right-click. The context menu shows the interpolation mode. To change the interpolation mode for a key, select the key, Right-click and select another interpolation mode.



The context menu shows the interpolation mode for the selected key. Use the context menu to change the interpolation mode.

Some interpolation modes break the left and right tangents so that they can be positioned separately. When tangents are broken, you can set a separate interpolation mode for the animation curve before the key and the animation curve after the key. For more details on the different interpolation modes, see Editing Curves. In the Animation window documentation, the interpolation mode is referred to as **tangent type**.

2017–08–10 Page published with limited editorial review

Deleting keys

Leave feedback

The **Curves view** provides the following methods for deleting keys:

Right-click on a key and select **Delete Key** from the context menu. This method does not affect selected keys.

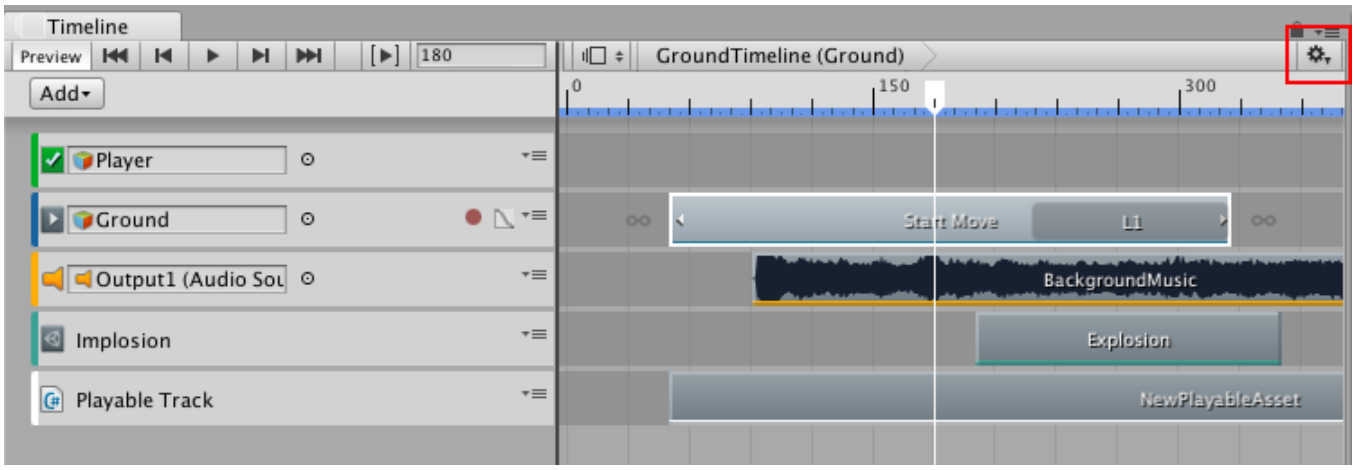
Select a key and either press Delete or Right-click and select **Delete Key** from the context menu.

2017–08–10 Page published with limited editorial review

Timeline Settings

Leave feedback

Use the **Timeline** Settings to set the unit of measurement for the **Timeline Asset**, to set the duration mode of the **Timeline Asset**, and to set the **Timeline** Editor window snap settings.



Click the Cog icon in the Timeline Editor window to view the Timeline Settings menu

Seconds or Frames

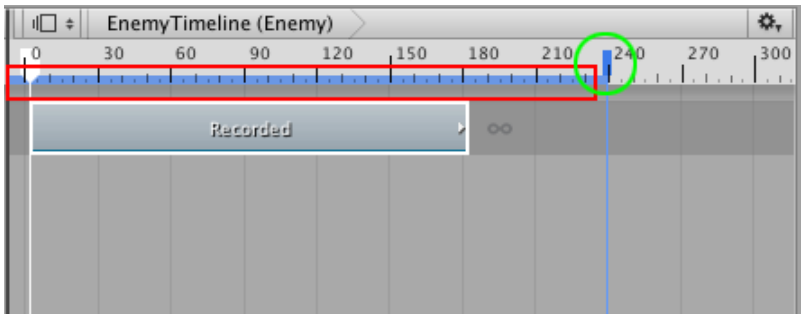
Select either Seconds or Frames to set the **Timeline Editor window** to display time in either seconds or frames.

Duration Mode

Use the Duration Mode to set whether the duration of the **Timeline Asset** extends to the end of the last clip (Based On Clips), or extends to a specific time or frame (Fixed Length). When the Duration Mode is set to Fixed Length, use one of the following methods to change the length of the Timeline Asset:

Select the Timeline Asset in the Project window and use the **Inspector** window to set the Duration in seconds or frames.

In the Timeline Editor window, drag the blue marker on the timeline. The blue marker indicates the end of the Timeline Asset. A blue line indicates the duration of the Timeline Asset.



Timeline Asset duration (red rectangle) and end marker (green circle)

Frame Rate

Select one of the options under frame rate to set the play speed of the Timeline Asset. The overall speed of the Timeline Asset accelerates or decelerates based on the number of **frames per second**. The higher the number of **frames per second**, the faster the entire timeline plays. The following frame rates are supported: Film (24 fps), PAL (25 fps), NTSC (29.97 fps), 30, 50, or 60.

Show Audio Waveforms

Enable **Show Audio Waveforms** to draw the waveforms for all **audio clips** on all audio tracks. For example, use an audio waveform as a guide when manually positioning an Audio clip of footsteps with the **Animation clip** of a humanoid walking. Disable **Show Audio Waveform** to hide audio waveforms. **Show Audio Waveforms** is enabled by default.

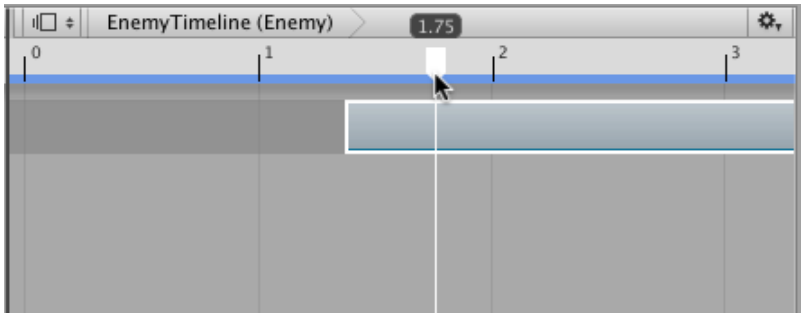
Snap to Frame

Enable **Snap to Frame** to manipulate clips, preview **Timeline instances**, drag the playhead, and position the playhead using frames. Disable **Snap to Frame** to use subframes. **Snap to Frame** is enabled by default.

For example, when **Snap to Frame** is disabled and you drag the **Timeline Playhead**, it moves the playhead between frames, the format of Playhead Location displays differently depending on whether the timeline is set to **Seconds** or **Frames**:

When the timeline is set to **Frames**, the **Playhead Location field** shows frames and subframes. For example, 8 frames and 34 subframes displays as 8.34.

When the timeline is set to **Seconds**, the Playhead Location field shows seconds, frames, and subframes. For example, 6 seconds, 17 frames, and 59 subframes displays as 6:17 [.59].



Disable Snap to Frame to position clips and drag the playhead between frames

Manipulating clips, previewing Timeline instances, and positioning the playhead at the subframes level is useful when attempting to synchronize animation and effects with audio. Many high-end audio processing software products create audio waveforms with subframe accuracy.

Edge Snap

Enable the **Edge Snap** option to snap clips during positioning, trimming, and creating blends. When enabled, clips are snapped when the start or end of a clip is dragged within 10 **pixels** of the start or end of a clip on another track, the start or end of a clip on the same track, the start or end of the entire timeline, or the playhead. Disable **Edge Snap** to create accurate blends, ease-ins, or ease-outs. **Edge Snap** is enabled by default.

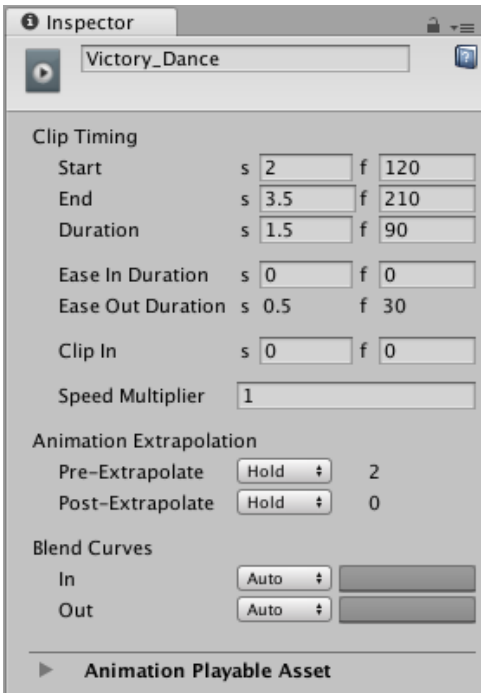
2017–12–07 Page amended with limited editorial review

Timeline and the Inspector Window

Leave feedback

The **Inspector** window displays information about the selected **GameObject** including all attached **components** and their properties. This section documents the properties in the **Inspector** window that appear when you select one or many **Timeline** objects: a **Timeline Asset**, a track, or a clip.

If you select a single Timeline object, the Inspector window displays the properties for the selected object. For example, if you select an **Animation clip**, the Inspector window shows the common properties and playable asset properties for the selected Animation clip.



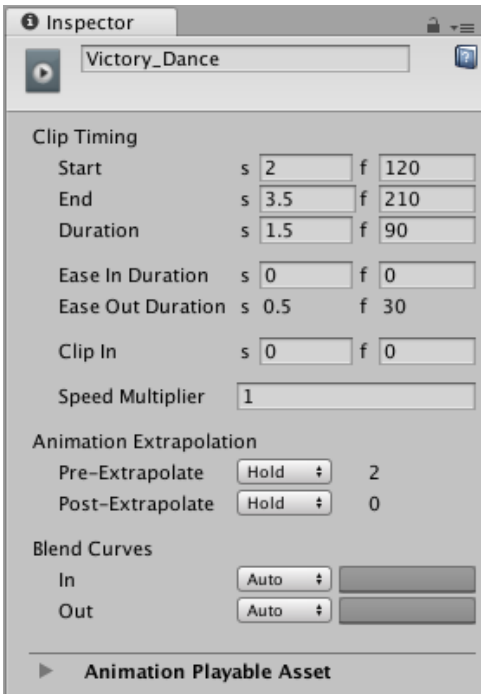
Inspector window when selecting an Animation clip in the Timeline Editor window

If you select multiple Timeline objects, and the selection includes Timeline objects with common properties, the Inspector window shows two sections: a section with properties that apply to the entire selection, and a section of common properties that apply to each selected object individually.

For example, if you select an **Audio clip** on one track and two Animation clips on another track, the Inspector window includes **Multiple Clip Timing** properties and **Clip Timing** properties:

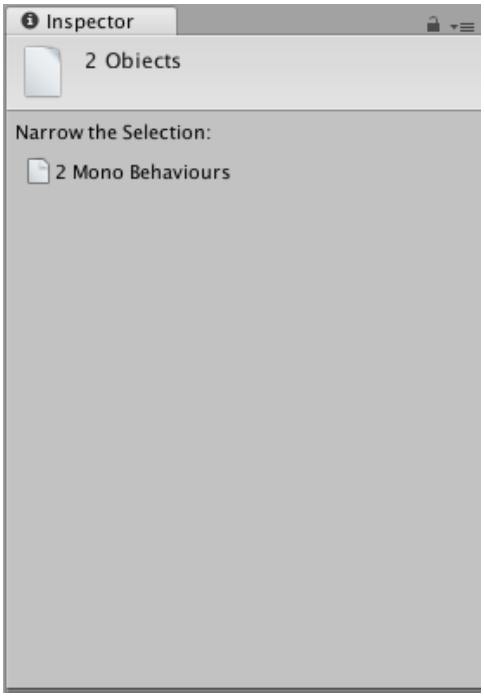
Use the **Multiple Clip Timing** properties to change the Start or End of the selection as a group. For example, if you change the Start to frame 30, the selection of clips start at frame 30. This moves the start of the first clip to frame 30 and the remaining selected clips are placed relative to the first clip, respecting gaps between selected clips.

Use the **Clip Timing** properties to change the common properties for each selected clip individually. For example, if you change the Ease In Duration to 10 frames, the Ease In Duration of each selected clip changes to 10 frames.



Inspector window when selecting multiple clips, on multiple tracks, in the Timeline Editor window. If the selected clips have different values for a property, the value is represented with a dash ("-")

If your selection includes Timeline objects that do not have common properties, the Inspector window prompts you to narrow the selection. For example, if you select an Animation track and an Audio clip in the Timeline Editor, you are prompted to narrow the selection.



If you select objects of different types, the Inspector window prompts you to narrow the selection to similar objects

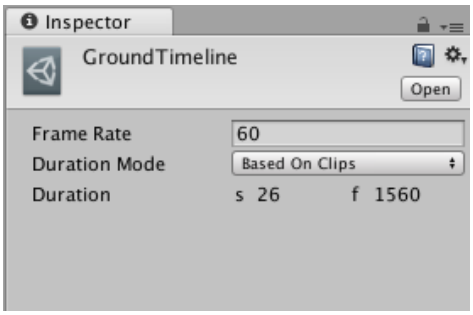
---

2017–12–07 Page amended with limited editorial review

Setting Timeline properties

Leave feedback

Use the **Inspector** window to set the Frame Rate, the duration mode, and a fixed length for the selected **Timeline Asset**. From the Project window, select a **Timeline Asset** to view its properties.



Inspector window when selecting a Timeline Asset in the Project.

The Timeline properties are also found in the Timeline Settings in the **Timeline Editor window**.

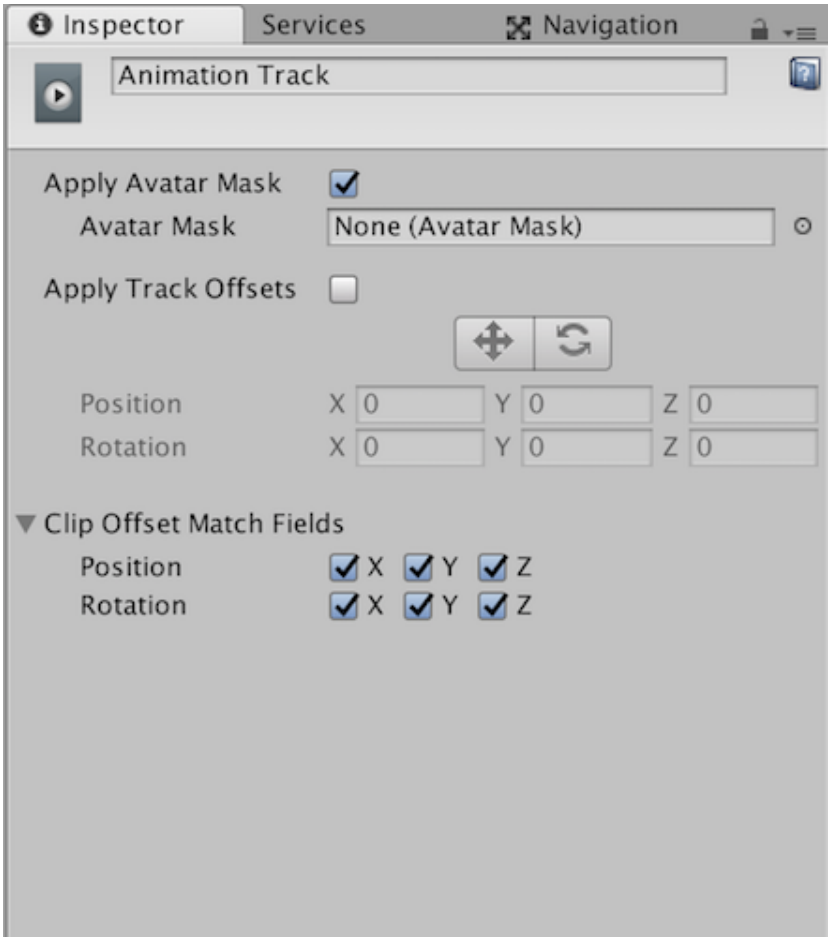
Property:	Function:
Frame Rate	Use Frame Rate to set the play speed of the **Timeline Asset**. The overall speed of the **Timeline Asset** accelerates or decelerates based on the number of **frames per second**. The higher the number of **frames per second**, the faster the entire timeline plays.
Duration Mode	Use Duration Mode to set whether the duration of the Timeline Asset is based on the clips in the timeline or a fixed length.
Based On Clips	Select Based On Clips to set the length of the Timeline Asset based on the end of the last clip. The Duration property shows the length of the Timeline Asset in seconds and frames.
Fixed Length	Select Fixed Length to use the Duration property to set the length of the Timeline Asset to a specific number of seconds or frames.
Duration	The Duration property displays the length of the Timeline Asset in seconds and frames. The Duration property is editable when the Duration Mode is set to Fixed Length.

2017–12–07 Page published with limited editorial review

Setting Track properties

Leave feedback

Use the **Inspector** window to change the name of a track and its properties. The available properties depend on the type of track selected. For example, select an Animation Track to apply an **avatar mask**, apply track offsets, and to specify which transforms are modified when performing match offsets between **Animation clips**.



Inspector window when selecting an Animation track in the Timeline Editor window

Not all tracks have properties. See the following dedicated sections for tracks with properties:

Activation Track properties

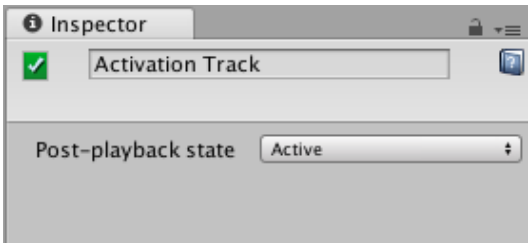
Animation Track properties

2017–12–07 Page published with limited editorial review

Activation Track properties

Leave feedback

Use the **Inspector** window to change the name of an Activation track and to set the state of its bound **GameObject** when the **Timeline Asset** finishes playing.



Inspector window when selecting an Activation track in the Timeline Editor window

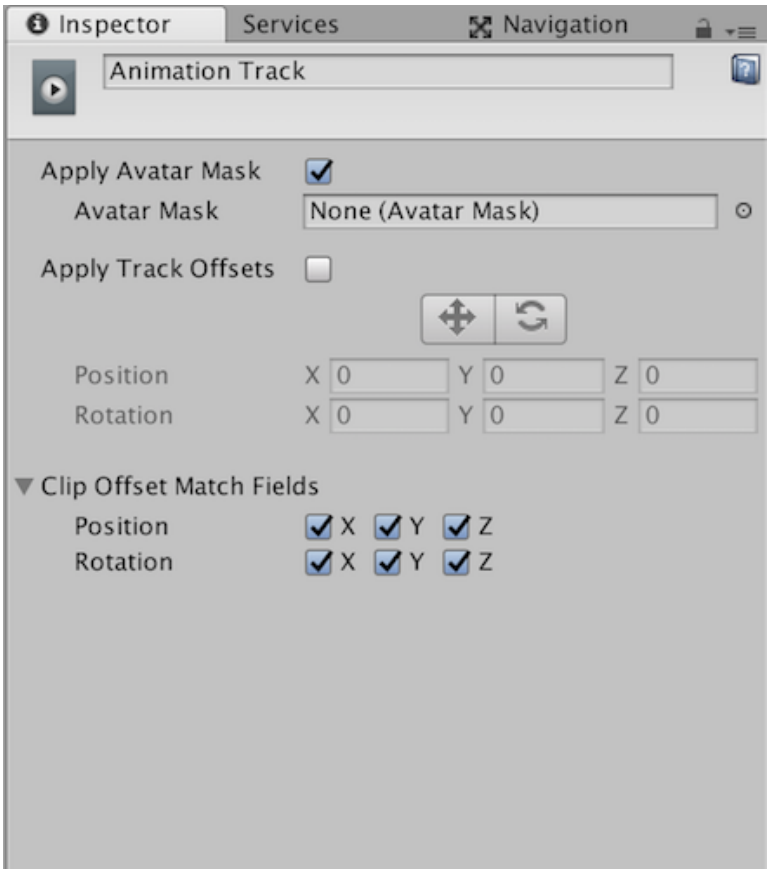
Property:	Function:
Display Name	The name of the Activation track shown in the **Timeline Editor window** and in the Playable Director **component**. The Display Name applies to the **Timeline Asset** and all of its Timeline instances. You can only modify the Display Name by selecting the Activation track while editing a **Timeline instance**.
Post-playback state	Use the Post-playback state to set the activation state for the bound GameObject when the Timeline Asset stops playing. The Post-playback state applies to the Timeline Asset and all of its Timeline instances.
Active	Select to activate the bound GameObject when the Timeline Asset finishes playing.
Inactive	Select to deactivate the bound GameObject when the Timeline Asset finishes playing.
Revert	Select to revert the bound GameObject to its activation state before the Timeline Asset began playing. For example, if the Timeline Asset ends with the GameObject set to inactive, and the GameObject was active before the Timeline Asset began playing, then the GameObject reverts to active.
Leave As Is	Select to set the activation state of the bound GameObject to same state at the end of the Timeline Asset. For example, if the Timeline Asset ends with the GameObject set to inactive, the GameObject remains inactive.

2017–12–07 Page published with limited editorial review

Animation Track properties

Leave feedback

Use the **Inspector** window to change the name of an Animation track, apply an **avatar mask**, apply track offsets, and to specify the transforms that are matched when matching clip offsets between **Animation clips**.



Inspector window when selecting an Animation track in the Timeline Editor window

Property:	Function:
Display Name	The name of the Animation track shown in the **Timeline** Editor window and in the Playable Director **component**. The Display Name applies to the **Timeline Asset** and all of its **Timeline** instances. You can only modify the Display Name by selecting the Animation track when editing a **Timeline instance**.
Apply Avatar__An interface for retargeting animation from one rig to another. More info See in Glossary Mask__	Use this property to enable **Avatar** masking. When enabled, the animation for all Animation clips on the track are applied based on the selected **Avatar** Mask.
Avatar Mask	Use this property to select the Avatar Mask applied to all Animation clips on the Animation track. An Avatar Mask defines which humanoid body parts are animated by Animation clips on the selected Animation track. The body parts that are masked are animated by other Animation tracks in the **Timeline Asset**. For example, you can use an Avatar Mask to combine the lower-body animation on an Animation track with the upper body animation on an Override Animation track.

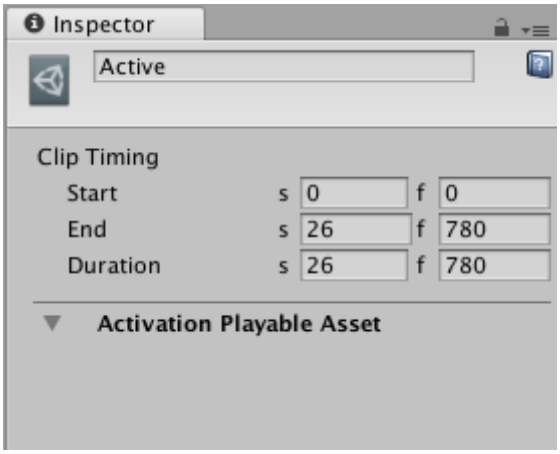
Property:	Function:
Apply Track Offsets	Enable Apply Track Offsets to apply the same position and rotation offsets to all Animation clips, on the selected Animation track. There are two methods for applying a track offset: set the position and rotation offsets with **gizmos** in the **Scene view**, or specify the exact position and rotation track offset.
Move tool	Enable the Move tool to show the Move Gizmo in the **Scene** view. Use the Move Gizmo to visually position the track offset. Positioning the Move Gizmo changes the Position properties.
Rotate tool	Enable the Rotate tool to show the Rotate Gizmo in the Scene view. Use the Rotate Gizmo to visually rotate the track offset. Rotating the Rotate Gizmo changes the Rotation properties.
Position	Use the Position properties to set the track offset in X, Y, and Z coordinates.
Rotation	Use the Rotation properties to set the track rotation offset around the X, Y, and Z axes.
Clip Offset Match Fields	Expand Clip Offset Match Fields to display a series of checkboxes that choose which transforms are matched when matching clip offsets between Animation clips. The Clip Offset Match Fields set the default matching options for all Animation clips on the same track. Use the Animation Clip playable asset properties to override these defaults for each Animation clip.

2017–12–07 Page published with limited editorial review

Setting Clip properties

Leave feedback

Use the **Inspector** window to change the name of a clip, its timing, its blend properties, and other clip properties. The available properties depend on the type of clip selected. For example, select an Activation clip to change its name and set its **Clip Timing**.



Inspector window when selecting an Activation clip in the Timeline Editor window

Not all clips have properties. See the following dedicated sections for clips with properties:

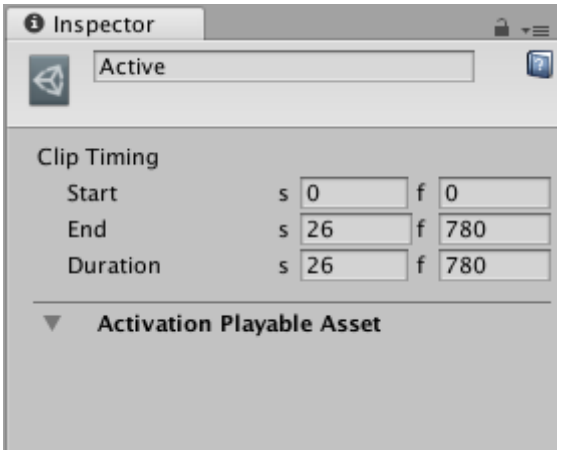
Activation Clip properties
Animation Clip common properties
Animation Clip playable asset properties
Audio Clip properties

2017–12–07 Page published with limited editorial review

Activation Clip properties

Leave feedback

Use the **Inspector** window to change the name of an Activation clip and its **Clip Timing**.



Inspector window when selecting an Activation clip in the Timeline Editor window

Display Name

The name of the Activation clip shown in the **Timeline** Editor window. By default, each Activation clip is named "Active".

Clip Timing properties

Use the **Clip Timing** properties to trim and change the duration of the Activation clip. Most of the timing properties are expressed in both seconds (s) and frames (f). When specifying seconds to modify a **Clip Timing** property, all decimal values are accepted. When specifying frames, only integer values are accepted. For example, if you attempt to enter 12.5 in a frames (f) field, it is set to 12 frames.

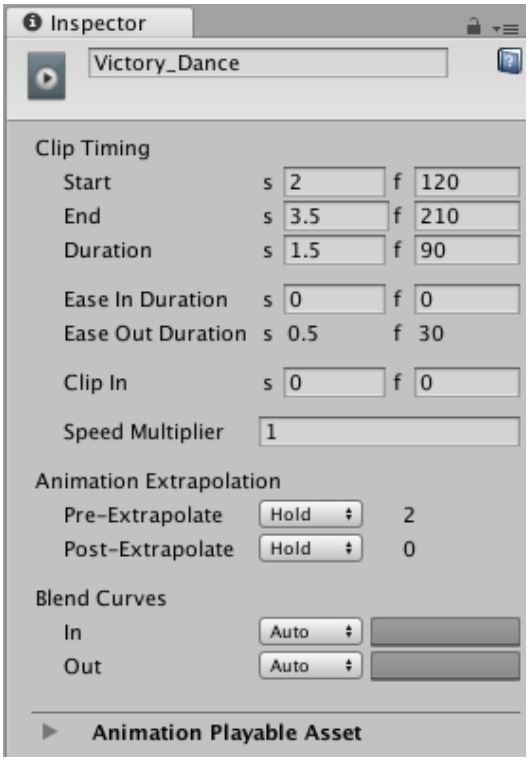
Property:	Function:
Start	The frame or time (in seconds) when the clip starts. Changing the Start property changes the position of the clip on its track in the **Timeline Asset**. Changing the Start may also affect the Duration. All clips use the Start property.
End	The frame or time (in seconds) when the clip ends. Changing the End property affects the Duration. All clips use the End property.
Duration	The duration of the clip in frames or seconds. Changing the Duration property also affects the End property. All clips use the Duration property.

2017–12–07 Page published with limited editorial review

Animation Clip common properties

Leave feedback

Use the **Inspector** window to change the common properties of an **Animation clip**. The common properties of an **Animation Clip** include its name, its timing, play speed, its blend properties, and its **extrapolation** settings.



Inspector window when selecting an Animation clip in the Timeline Editor window

Display Name

The name of the Animation clip shown in the **Timeline** Editor window.

Clip Timing properties

Use the **Clip Timing** properties to trim, change the duration, change the ease-in and ease-out duration or extrapolation, and to adjust the play speed of the Animation clip.

Most of the following timing properties are expressed in both seconds (s) and frames (f). When specifying seconds to modify a **Clip Timing** property, all decimal values are accepted. When specifying frames, only integer values are accepted. For example, if you attempt to enter 12.5 in a frames (f) field, it is set to 12 frames.

Property:	Function:
Start	The frame or time (in seconds) when the clip starts. Changing the Start property changes the position of the clip on its track in the **Timeline Asset**. Changing the Start may also affect the Duration.
End	The frame or time (in seconds) when the clip ends. Changing the End property affects the Duration. All clips use the End property.
Duration	The duration of the clip in frames or seconds. Changing the Duration property also affects the End property.
Ease In Duration	Sets the number of seconds or frames that it takes for the clip to ease in. If the beginning of the clip overlaps and blends with another clip, the Ease In Duration cannot be edited and instead shows the duration of the blend between clips. See Blending clips.

Property:	Function:
Ease Out Duration	Sets the number of seconds or frames that it takes for the clip to ease out. If the end of the clip overlaps and blends with another clip, the Ease Out Duration cannot be edited and instead shows the duration of the blend between clips. In this case, trim or position the clip to change the duration of the blend between clips. See Blending clips.
Clip In	Sets the offset of when the source clip should start playing. For example, to play the last 10 seconds of a 30 second **audio clip**, set Clip In to 20 seconds.
Speed Multiplier	A multiplier on the playback speed of the clip. This value must be greater than 0. Changing this value of the clip will change the duration of the clip to play the same content.

Animation Extrapolation

Use the **Animation Extrapolation** properties to set the **gap extrapolation** before and after an Animation clip. The term **gap extrapolation** refers to how an Animation track approximates or extends animation data in the gaps before, between, and after the Animation clips on a track.

Only Animation clips use the **Animation Extrapolation** properties. There are two properties for setting the gap extrapolation between Animation clips.

Property:	Function:
Pre-Extrapolate	Controls how animation data is approximated in the gap before an Animation clip. The Pre-Extrapolate property affects easing-in an Animation clip.
Post-Extrapolate	Controls how animation data extends in the gap after an Animation clip. The Post-Extrapolate property affects easing-out an Animation clip.

Blend Curves

Use the **Blend Curves** to customize the transition between the **out-going clip** and the **incoming clip** when blending between two Animation clips. See Blending clips for details on how to blend clips and how to customize blend curves.

When easing-in or easing-out clips, the **Blend Curves** allow you to customize the curve that eases-in a clip and the curve that eases-out a clip.

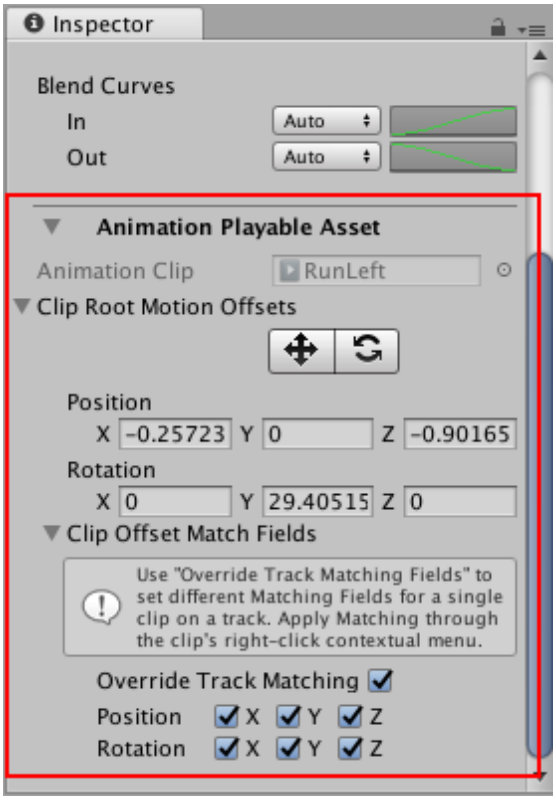
2017–12–07 Page published with limited editorial review

Animation Clip playable asset properties

Leave feedback

Use the **Inspector** window to change the playable **asset** properties of an **Animation clip**. The playable **asset** properties include controls for manually transforming the **root motion** offset of the **Animation clip** and options for overriding default clip matching.

To view the playable asset Animation Clip properties, select an Animation clip in the **Timeline** Editor window and expand **Animation Playable Asset** in the Inspector window.



Inspector window showing the playable asset properties of the selected Animation clip

Clip Root Motion Offsets

Use the **Clip Root Motion Offsets** to apply position and rotation offsets to the root motion of the selected Animation clip. There are two methods of applying clip root offsets:

Match the clip offsets to set the root motion offsets based on the end of the previous Animation clip, or the start of the next Animation clip. What gets matched depends on the **Clip Offset Match Fields**.

Use the tools and properties under **Clip Root Motion Offsets** to manually set the position and rotation of the root motion offsets.

Property:	Function:
Move tool	Enable the Move tool to show a Move **Gizmo** in the **Scene view**. Use the Move **Gizmo** to manually position the root motion offset for the selected Animation clip. Using the Move **Gizmo** changes the Position properties.

Property:	Function:
Rotate tool	Enable the Rotate tool to show a Rotate Gizmo in the **Scene** view. Use the Rotate Gizmo to manually rotate the root motion offset for the selected Animation clip. Using the Rotate Gizmo changes the Rotation properties.
Position	Use the Position properties to manually set the clip offset in X, Y, and Z coordinates. By default, the Position coordinates are set to zero and are relative to the track offsets.
Rotation	Use the Rotation properties to manually set the clip rotation offset around the X, Y, and Z axes. By default, the Rotation coordinates are set to zero and are relative to the track offsets.

Clip Offset Match Fields

Use the **Clip Offset Match Fields** to choose which transforms are matched when matching clip offsets. By default, **Override Track Match Fields** is disabled and the **Clip Offset Match Fields** show and use the matching options set at the Animation track level.

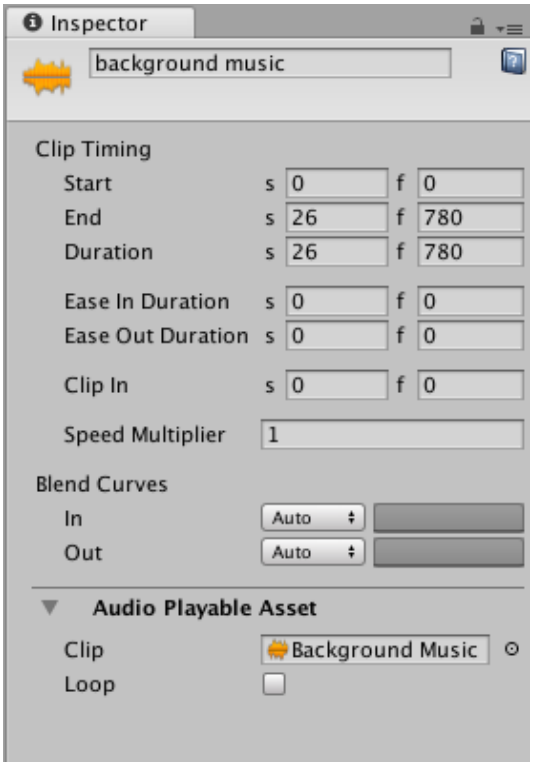
Enable **Override Track Match Fields** to override the track match options and set which transformations are matched for the selected Animation clip.

2017–12–07 Page published with limited editorial review

Audio Clip properties

Leave feedback

Use the **Inspector** window to change the properties of an **Audio clip**. These properties include the name of the clip, its timing, play speed, its blend properties, the audio file used by the **Audio clip**, and whether **Audio clip** loops or plays once.



Inspector window when selecting an Audio clip in the Timeline Editor window

Display Name

The name of the Audio clip shown in the **Timeline** Editor window. This is not the name of the audio file that is used for the waveform. The name of the audio file is part of the **Audio Playable Asset** properties.

Clip Timing properties

Use the **Clip Timing** properties to trim and change the duration of the Audio clip. Most of the timing properties are expressed in both seconds (s) and frames (f). When specifying seconds to modify a **Clip Timing** property, all decimal values are accepted. When specifying frames, only integer values are accepted. For example, if you attempt to enter 12.5 in a frames (f) field, it is set to 12 frames.

Property:	Function:
Start	The frame or time (in seconds) when the clip starts. Changing the Start property changes the position of the clip on its track in the **Timeline Asset**. Changing the Start may also affect the Duration. All clips use the Start property.
End	The frame or time (in seconds) when the clip ends. Changing the End property affects the Duration. All clips use the End property.
Duration	The duration of the clip in frames or seconds. Changing the Duration property also affects the End property. All clips use the Duration property.

Blend Curves

Use the **Blend Curves** to customize the transition between the **out-going clip** and the **incoming clip** when blending between two Audio clips. See Blending clips for details on how to blend clips and how to customize blend curves.

When easing-in or easing-out clips, the **Blend Curves** allow you to customize the curve that eases-in an Audio clip and the curve that eases-out an Audio clip. See Easing-in and Easing-out clips for details.

Audio Playable Asset properties

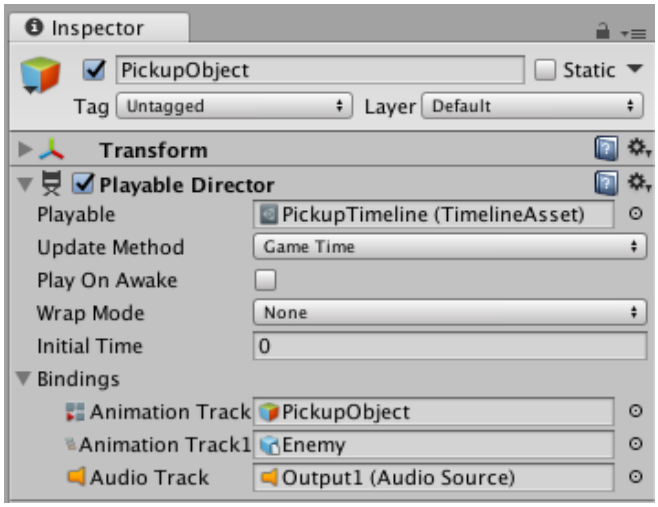
Use the **Audio Playable Asset** properties to select the Audio file used by the Audio clip and to set whether the selected Audio clip loops (**Loop** enabled) or plays once (**Loop** disabled).

2017–12–07 Page published with limited editorial review

Playable Director component

Leave feedback

The Playable Director **component** stores the link between a **Timeline instance** and a **Timeline Asset**. The Playable Director **component** controls when the **Timeline instance** plays, how the **Timeline instance** updates its clock, and what happens when the **Timeline instance** finishes playing.



Playable Director component added to the GameObject named PickupObject. The GameObject is associated with the PickupTimeline Timeline Asset.

The Playable Director component also shows the list of tracks, from the associated **Timeline Asset** (**Playable** property), that animate **GameObjects** in the **scene**. The link between Timeline Asset tracks and **GameObjects** in the **scene** is referred to as **binding** or **Track binding**. See Timeline Overview section for details on binding and the relationship between Timeline Assets and Timeline instances.

Playable

Use the **Playable** property to manually associate a Timeline Asset with a GameObject in the scene. When you make this association, you create a Timeline instance for the selected Timeline Asset. After you create a Timeline instance, you can use the other properties in the Playable Director component to control the instance and to choose which GameObjects in the scene are animated by the Timeline Asset.

Update Method

Use the Update Method to set the clock source that the Timeline instance uses to update its timing. The Update Method supports the following clock sources:

DSP: Select for sample accurate audio scheduling. When selected, the Timeline instance uses the same clock source that processes audio. DSP stands for digital signal processing.

Game Time: Select to use the same clock source as the game clock. This clock source is affected by time scaling.

Unscaled Game Time: Select to use the same clock source as the game clock, but without being affected by time scaling.

Manual: Select to not use a clock source and to manually set the clock time through scripting.

Play on Awake

Whether the Timeline instance is played when game play is initiated. By default, a Timeline instance is set to begin as soon as the scene begins playback. To disable the default behaviour, disable the **Play on Awake** option in the Playable Director component.

Wrap Mode

The behaviour when the Timeline instance ends playback. The Wrap mode also defines the behaviour when the **Timeline Editor window** is in Play Range mode. The following Wrap modes are supported:

Hold: Plays the Timeline instance once and holds on the last frame until playback is interrupted.

Loop: Plays the sequence repeatedly until playback is interrupted.

None: Plays the sequence once and then resets all animated properties to the values they held before playback.

Initial Time

The time (in seconds) at which the Timeline instance begins playing. The Initial Time adds a delay in seconds from when the Timeline instance is triggered to when playback actually begins. For example, if Play On Awake is enabled and Initial Time is set to five seconds, clicking the Play button in the Unity **Toolbar** starts Play mode and the Timeline instance begins at five seconds.

This is useful when you are working on a long cinematic and you want to preview the last few seconds of your Timeline instance.

Current Time

Use the Current Time field to view the progression of time according to the Timeline instance in the Timeline Editor window. The Current Time field matches the **Playhead Location field**. The Current Time field is useful when the Timeline Editor window is hidden. The Current Time field appears in the Playable Director Component when in **Timeline Playback mode** or when Unity is in Game Mode.

Bindings

Use the **Bindings** area to link GameObjects in the scene with tracks from the associated Timeline Asset (**Playable** property). When you link a GameObject to a track, the track animates the GameObject in the scene. The link between a GameObject and a track is referred to as **binding** or **Track binding**.

The **Bindings** area is split into two columns:

The first column lists the tracks from the Timeline Asset. Each track is identified by an icon and its track type.

The second column lists the GameObject linked (or **bound**) to each track.

The **Bindings** area does not list **Track groups**, Track sub-groups, or tracks that do not animate GameObjects. The Timeline Editor window shows the same bindings in the Track list.

2017–08–10 Page published with limited editorial review

Timeline glossary

Leave feedback

This section provides an alphabetical list of the terminology used throughout the Timeline documentation.

animatable property: A property belonging to a **GameObject**, or belonging to a **component** added to a **GameObject**, that can have different values over time.

animation: The result of adding two different keys, at two different times, for the same animatable property.

animation curve: The curve drawn between keys set for the same animatable property, at different frames or seconds. The position of the tangents and the selected interpolation mode for each key determines the shape of the animation curve.

binding or **Track binding**: Refers to the link between Timeline **Asset** tracks and the GameObjects in the **scene**. When you link a GameObject to a track, the track animates the GameObject. Bindings are stored as part of the Timeline instance.

blend and **blend area**: The area where two **Animation clips**, **Audio clips**, or Control clips overlap. The overlap creates a transition that is referred to as a **blend**. The duration of the overlap is referred to as the **blend area**. The blend area sets the duration of the transition.

Blend In curve: In a blend between two Animation clips, Audio clips, or Control clips, there are two blend curves. The blend curve for the incoming clip is referred to as the **Blend In** curve.

Blend Out curve: In a blend between two Animation clips, Audio clips, or Control clips, there are two blend curves. The blend curve for the out-going clip is referred to as the **Blend Out curve**.

clip: A generic term that refers to any clip within the Clips view of the Timeline Editor window.

Clips view: The area in the Timeline Editor window where you add, position, and manipulate clips.

Control/Command: This term is used when instructing the user to press or hold down the Control key on Windows, or the Command key on Mac.

Curves view: The area in the Timeline Editor window that shows the animation curves for Infinite clips or for Animation clips that have been converted from Infinite clips. The Curves view is similar to Curves mode in the Animation window.

Gap extrapolation: How an Animation track approximates animation data in the gaps before and after an Animation clip.

field: A generic term that describes an editable box that the user clicks and types-in a value. A field is also referred to as a **property**.

incoming clip: The second clip in a blend between two clips. The first clip, the out-going clip, transitions to the second clip, the **incoming clip**.

infinite clip: A special animation clip that contains basic key animation recorded directly to an Animation track within the Timeline Editor window. An Infinite clip cannot be positioned, trimmed, or split because it does not have a defined duration: it spans the entirety of an Animation track.

interpolation: The estimation of values that determine the shape of an animation curve between two keys.

interpolation mode: The interpolation algorithm that draws the animation curve between two keys. The interpolation mode also joins or breaks left and right tangents.

key: The value of an animatable property, set at a specific point in time. Setting at least two keys for the same property creates an animation.

out-going clip: The first clip in a blend between two clips. The first clip, the **out-going clip**, transitions to the second clip, the incoming clip.

Playhead Location field: The field that expresses the location of the Timeline Playhead in either frames or seconds, depending on the Timeline Settings.

property: A generic term for the editable fields, buttons, checkboxes, or menus that comprise a component. An editable field is also referred to as a **field**.

tangent: One of two handles that controls the shape of the animation curve before and after a key. Tangents appear when a key is selected in the Curves view, or when a key is selected in the Curve Editor.

tangent mode: The selected interpolation mode used by the left tangent, right tangent, or both tangents.

Timeline or **Unity's Timeline**: Generic terms that refer to all features, windows, editors, and components related to creating, modifying, or reusing cut-scenes, cinematics, and game-play sequences.

Timeline Asset: Refers to the tracks, clips, and recorded animation that comprise a cinematic, cut-scene, game-play sequence, or other effect created with the Timeline Editor window. A Timeline Asset does not include bindings to the GameObjects animated by the Timeline Asset. The bindings to scene GameObjects are stored in the Timeline instance. The Timeline Asset is project-based.

Timeline Editor window: The official name of the window where you create, modify, and preview a Timeline instance. Modifications to a Timeline instance also affects the Timeline Asset.

Timeline instance: Refers to the link between a Timeline Asset and the GameObjects that the Timeline Asset animates in the scene. You create a Timeline instance by associating a Timeline Asset to a GameObject through a Playable Director component. The Timeline instance is scene-based.

Timeline Playback Controls: The row of buttons and fields in the Timeline Editor window that controls playback of the Timeline instance. The Timeline Playback Controls affect the location of the Timeline Playhead.

Timeline Playback mode: The mode that previews the Timeline instance in the Timeline Editor window. Timeline Playback mode is a simulation of Play mode. Timeline Playback mode does not support audio playback.

Timeline Playhead: The white marker and line that indicates the exact point in time being previewed in the Timeline Editor window.

Timeline Selector: The name of the menu in the Timeline Editor window that selects the Timeline instance to be previewed or modified.

track: A generic term that refers to any track within the Track list of the Timeline Editor window.

Track groups: The term for a series of tracks organized in an expandable and collapse collection of tracks.

Track list: The area in the Timeline Editor window where you add, group, and modify tracks.

2017–08–10 Page published with limited editorial review

UI

Leave feedback

The **UI** system allows you to create user interfaces fast and intuitively. This is an introduction to the major features of Unity's **UI** system.

Related tutorials: User Interface (UI)

Search the Unity Knowledge Base for tips, tricks and troubleshooting.

Canvas

Leave feedback

The **Canvas** is the area that all **UI** elements should be inside. The Canvas is a Game Object with a Canvas **component** on it, and all **UI** elements must be children of such a Canvas.

Creating a new UI element, such as an Image using the menu **GameObject > UI > Image**, automatically creates a Canvas, if there isn't already a Canvas in the scene. The UI element is created as a child to this Canvas.

The Canvas area is shown as a rectangle in the **Scene View**. This makes it easy to position UI elements without needing to have the Game View visible at all times.

Canvas uses the EventSystem object to help the Messaging System.

Draw order of elements

UI elements in the Canvas are drawn in the same order they appear in the Hierarchy. The first child is drawn first, the second child next, and so on. If two UI elements overlap, the later one will appear on top of the earlier one.

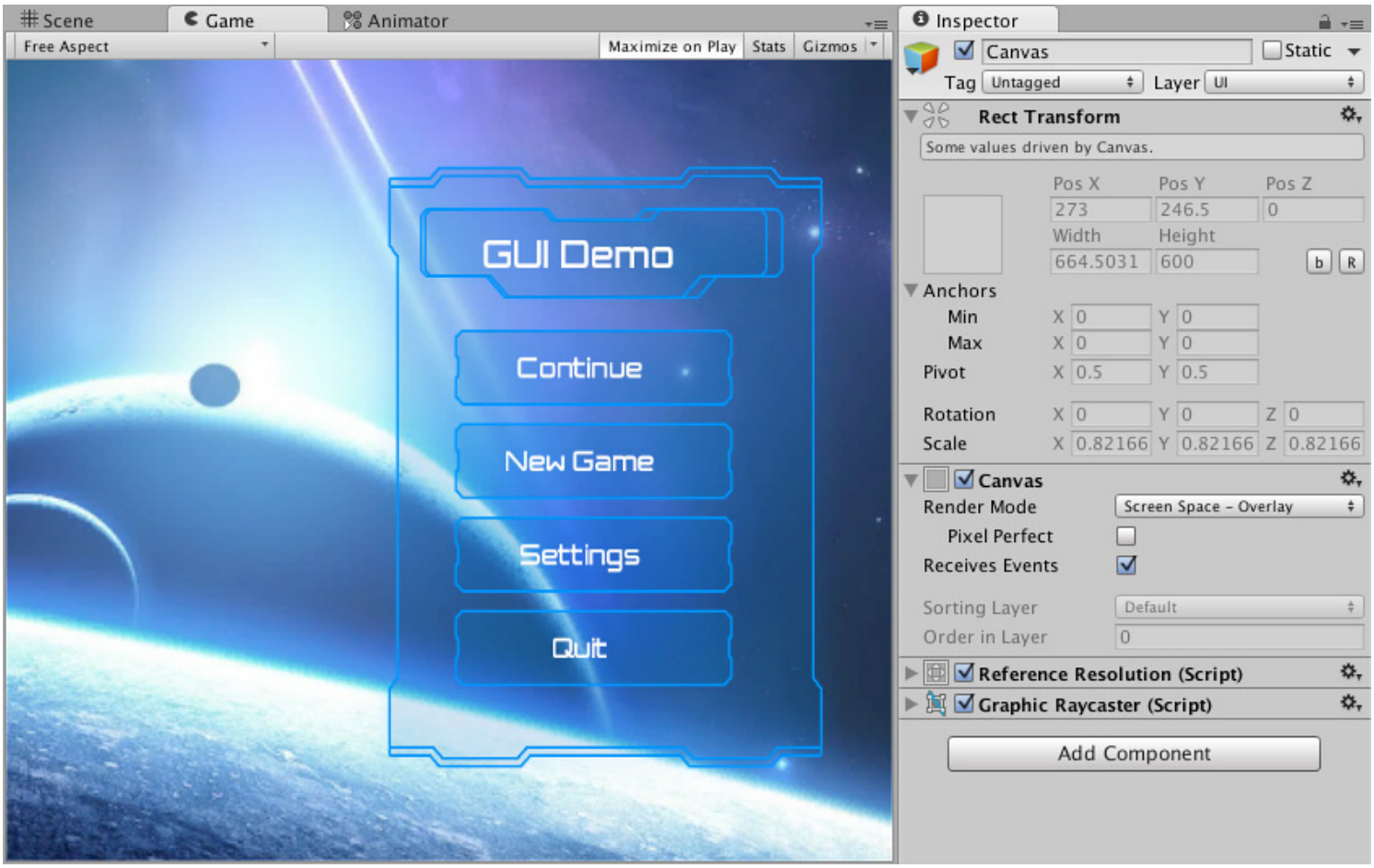
To change which element appear on top of other elements, simply reorder the elements in the Hierarchy by dragging them. The order can also be controlled from scripting by using these methods on the **Transform component**: SetAsFirstSibling, SetAsLastSibling, and SetSiblingIndex.

Render Modes

The Canvas has a **Render Mode** setting which can be used to make it render in screen space or world space.

Screen Space - Overlay

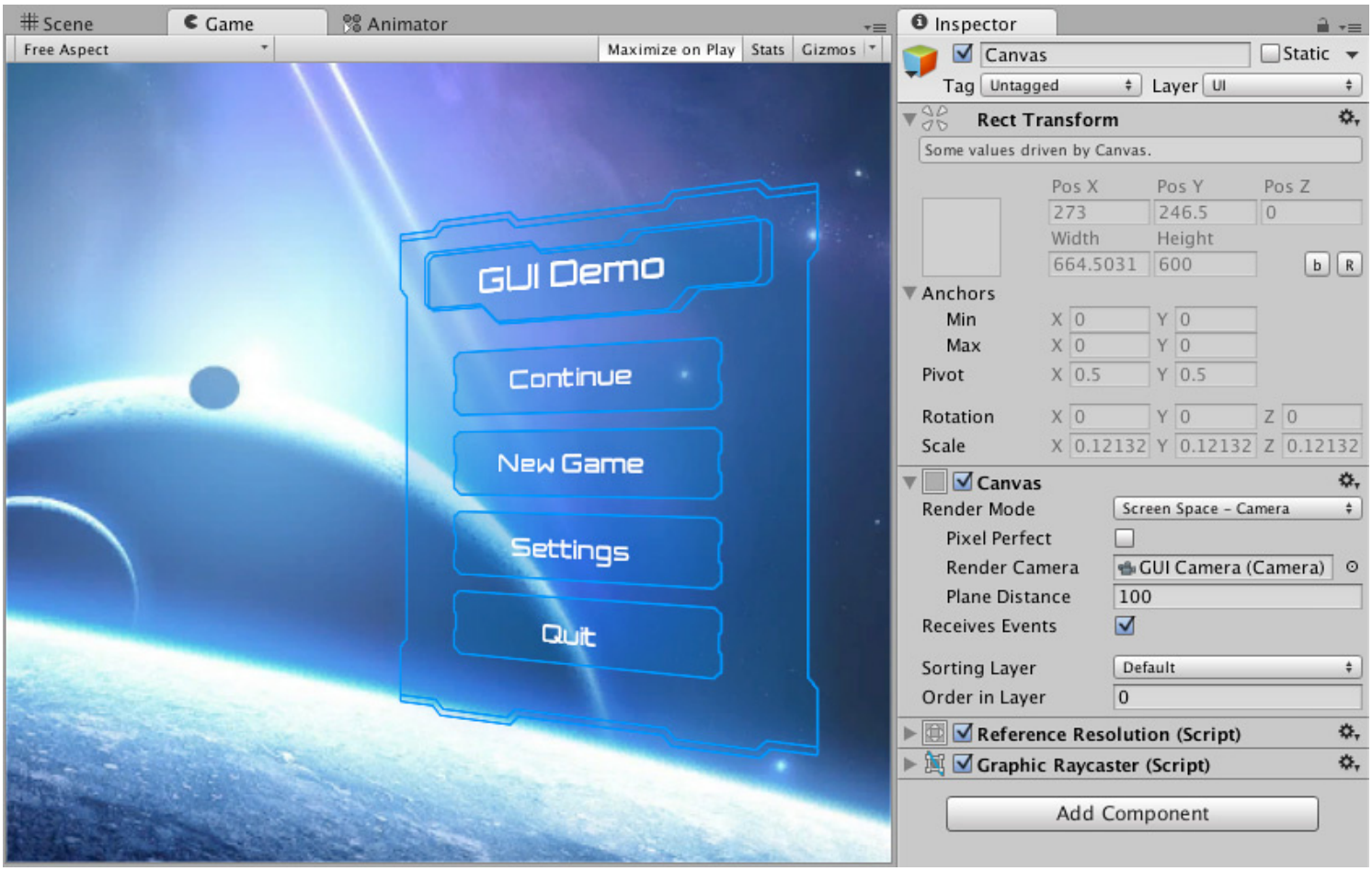
This render mode places UI elements on the screen rendered on top of the **scene**. If the screen is resized or changes resolution, the Canvas will automatically change size to match this.



UI in screen space overlay canvas

Screen Space - Camera

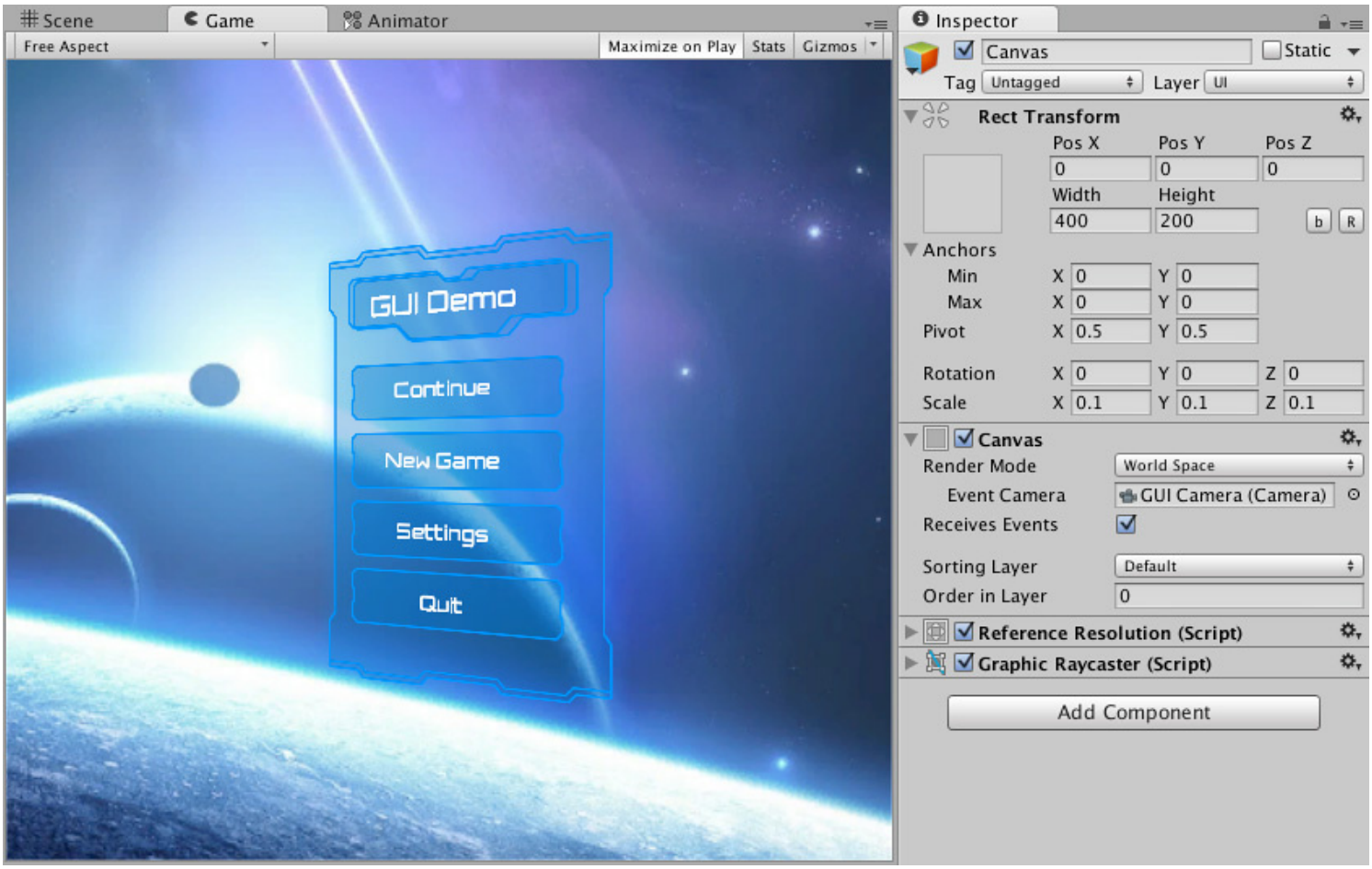
This is similar to **Screen Space - Overlay**, but in this render mode the Canvas is placed a given distance in front of a specified **Camera**. The UI elements are rendered by this camera, which means that the Camera settings affect the appearance of the UI. If the Camera is set to **Perspective**, the UI elements will be rendered with perspective, and the amount of perspective distortion can be controlled by the Camera **Field of View**. If the screen is resized, changes resolution, or the camera frustum changes, the Canvas will automatically change size to match as well.



UI in screen space camera canvas

World Space

In this render mode, the Canvas will behave as any other object in the scene. The size of the Canvas can be set manually using its Rect Transform, and UI elements will render in front of or behind other objects in the scene based on 3D placement. This is useful for UIs that are meant to be a part of the world. This is also known as a “diegetic interface”.



UI in world space canvas

Basic Layout

Leave feedback

In this section we'll look at how you can position UI elements relative to the Canvas and each other. If you want to test yourself while reading, you can create an Image using the menu **GameObject -> UI -> Image**.

The Rect Tool

Every **UI** element is represented as a rectangle for layout purposes. This rectangle can be manipulated in the **Scene View** using the **Rect Tool** in the **toolbar**. The Rect Tool is used both for Unity's 2D features and for **UI**, and in fact can be used even for **3D objects** as well.

Toolbar buttons with Rect Tool selected

The Rect Tool can be used to move, resize and rotate UI elements. Once you have selected a UI element, you can move it by clicking anywhere inside the rectangle and dragging. You can resize it by clicking on the edges or corners and dragging. The element can be rotated by hovering the cursor slightly away from the corners until the mouse cursor looks like a rotation symbol. You can then click and drag in either direction to rotate.

Just like the other tools, the Rect Tool uses the current pivot mode and space, set in the toolbar. When working with UI it's usually a good idea to keep those set to **Pivot** and **Local**.

Toolbar buttons set to Pivot and Local

Rect Transform

The **Rect Transform** is a new transform **component** that is used for all UI elements instead of the regular **Transform component**.

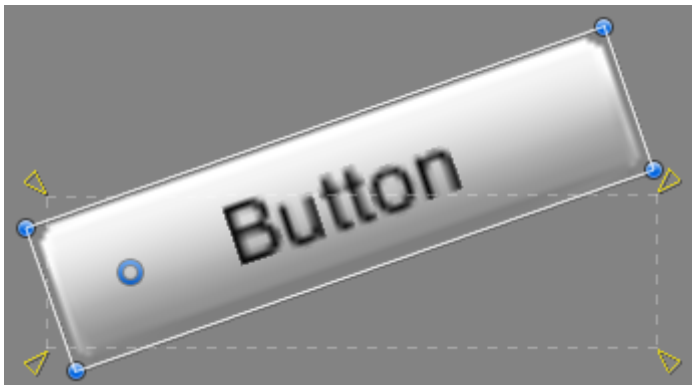
Rect Transforms have position, rotation, and scale just like regular Transforms, but it also has a width and height, used to specify the dimensions of the rectangle.

Resizing Versus Scaling

When the Rect Tool is used to change the size of an object, normally for **Sprites** in the 2D system and for 3D objects it will change the local *scale* of the object. However, when it's used on an object with a Rect Transform on it, it will instead change the width and the height, keeping the local scale unchanged. This resizing will not affect font sizes, border on sliced images, and so on.

Pivot

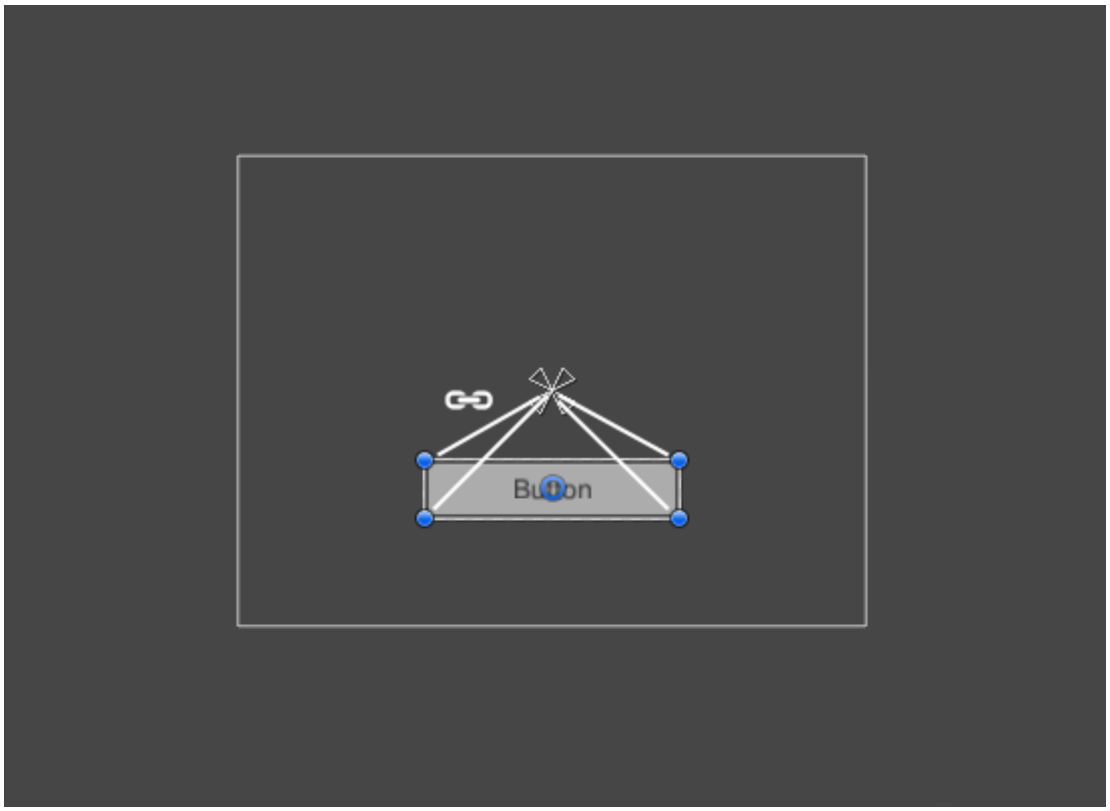
Rotations, size, and scale modifications occur around the pivot so the position of the pivot affects the outcome of a rotation, resizing, or scaling. When the toolbar Pivot button is set to Pivot mode, the pivot of a Rect Transform can be moved in the **Scene** View.



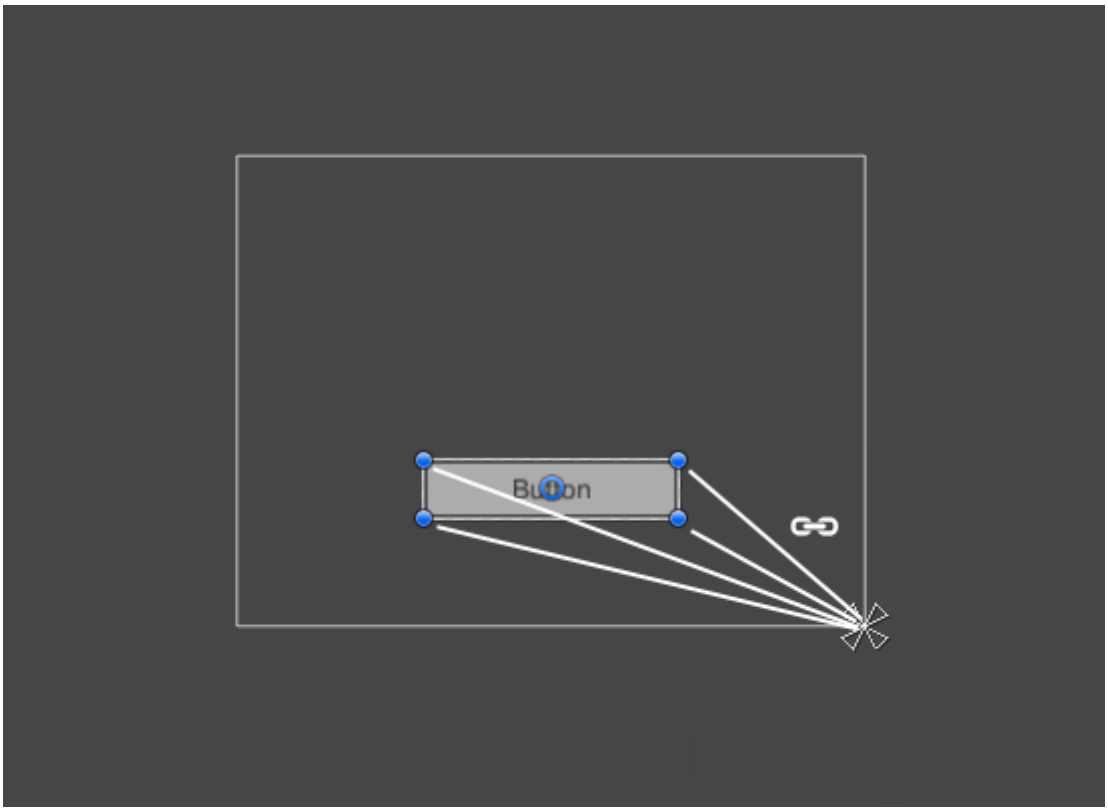
Anchors

Rect Transforms include a layout concept called **anchors**. Anchors are shown as four small triangular handles in the Scene View and anchor information is also shown in the **Inspector**.

If the parent of a Rect Transform is also a Rect Transform, the child Rect Transform can be anchored to the parent Rect Transform in various ways. For example, the child can be anchored to the center of the parent, or to one of the corners.

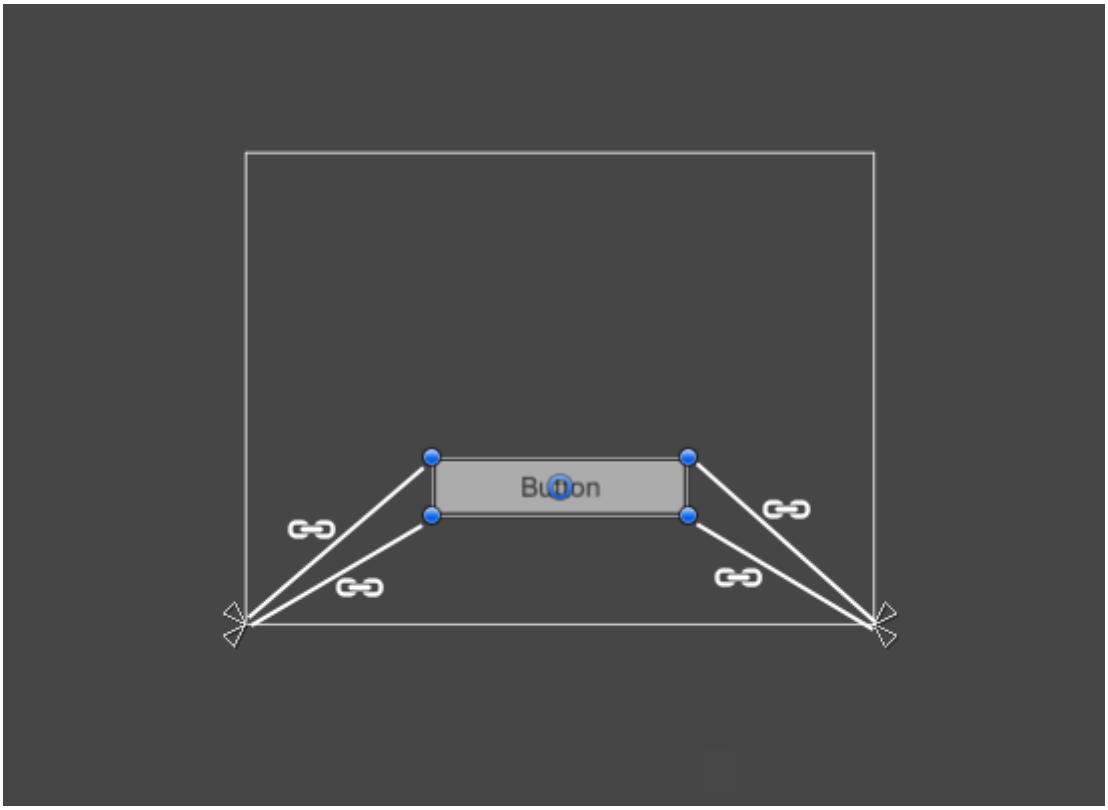


UI element anchored to the center of the parent. The element maintains a fixed offset to the center.



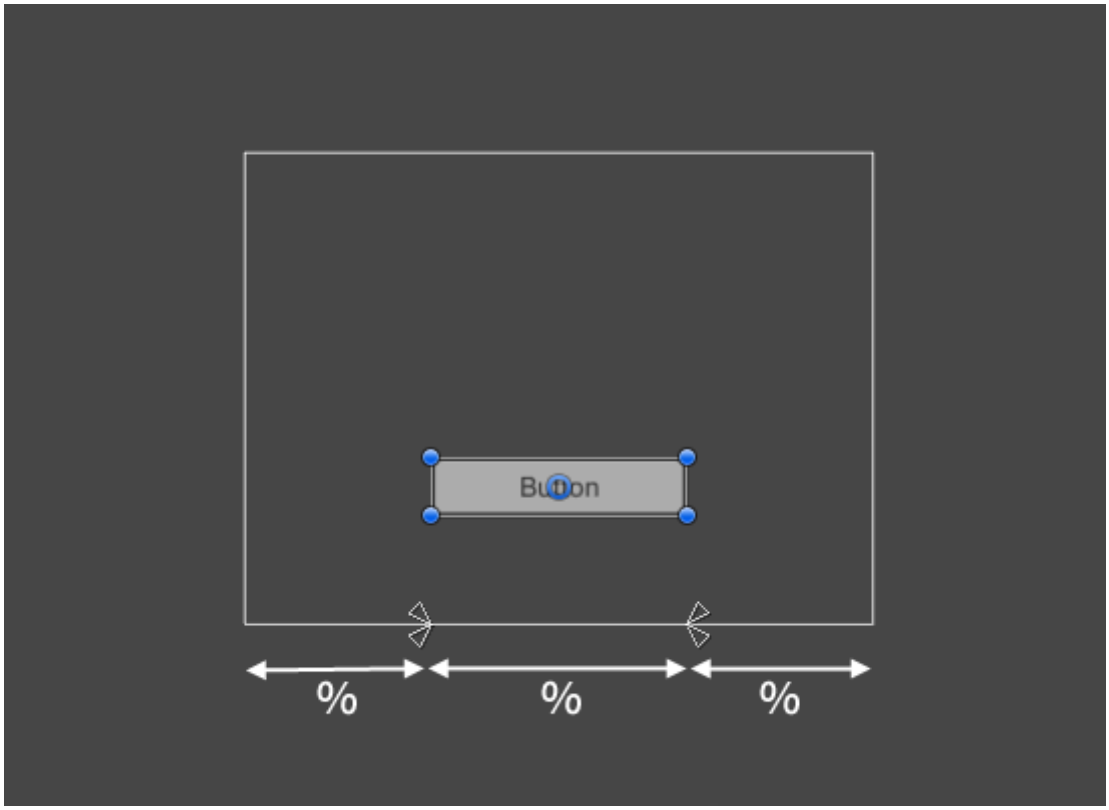
UI element anchored to the lower right corner of the parent. The element maintains a fixed offset to the lower right corner.

The anchoring also allows the child to stretch together with the width or height of the parent. Each corner of the rectangle has a fixed offset to its corresponding anchor, i.e. the top left corner of the rectangle has a fixed offset to the top left anchor, **etc**. This way the different corners of the rectangle can be anchored to different points in the parent rectangle.



UI element with left corners anchored to lower left corner of the parent and right corners anchored to lower right. The corners of the element maintains fixed offsets to their respective anchors.

The positions of the anchors are defined in fractions (or percentages) of the parent rectangle width and height. 0.0 (0%) corresponds to the left or bottom side, 0.5 (50%) to the middle, and 1.0 (100%) to the right or top side. But anchors are not limited to the sides and middle; they can be anchored to any point within the parent rectangle.



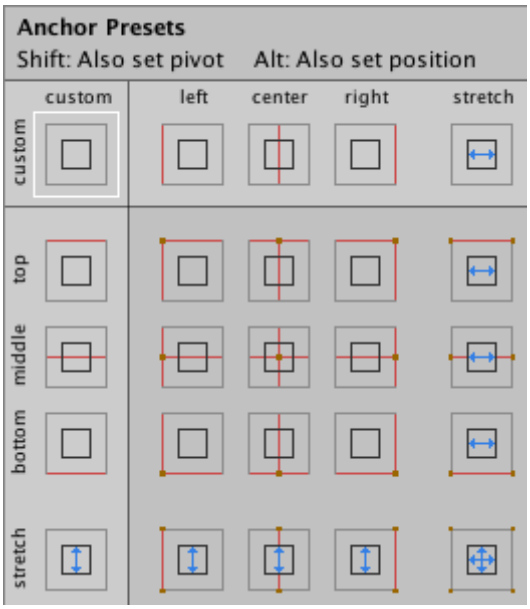
UI element with left corners anchored to a point a certain percentage from the left side of the parent and right corners anchored to a point a certain percentage from the right side of the parent rectangle.

You can drag each of the anchors individually, or if they are together, you can drag them together by clicking in the middle in between them and dragging. If you hold down **Shift** key while dragging an anchor, the corresponding corner of the rectangle will move together with the anchor.

A useful feature of the anchor handles is that they automatically snap to the anchors of sibling rectangles to allow for precise positioning.

Anchor presets

In the Inspector, the **Anchor Preset** button can be found in the upper left corner of the Rect **Transform component**. Clicking the button brings up the Anchor Presets dropdown. From here you can quickly select from some of the most common anchoring options. You can anchor the UI element to the sides or middle of the parent, or stretch together with the parent size. The horizontal and vertical anchoring is independent.

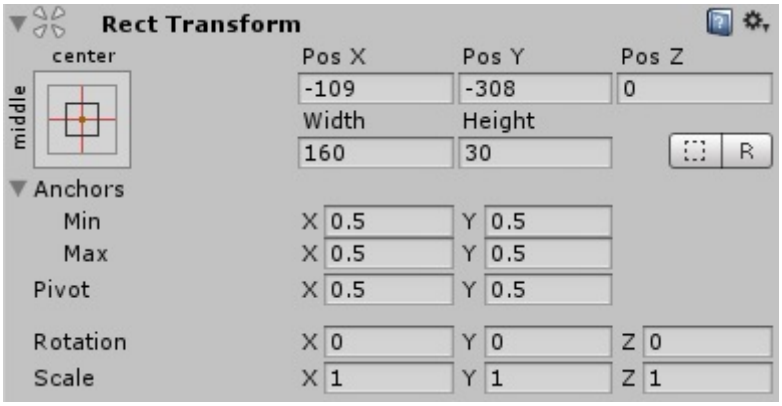


The Anchor Presets buttons displays the currently selected preset option if there is one. If the anchors on either the horizontal or vertical axis are set to different positions than any of the presets, the custom options is shown.

Anchor and position fields in the Inspector

You can click the Anchors expansion arrow to reveal the anchor number fields if they are not already visible. **Anchor Min** corresponds to the lower left anchor handle in the Scene View, and **Anchor Max** corresponds to the upper right handle.

The position fields of rectangle are shown differently depending on whether the anchors are together (which produces a fixed width and height) or separated (which causes the rectangle to stretch together with the parent rectangle).



When all the anchor handles are together the fields displayed are Pos X, Pos Y, Width and Height. The Pos X and Pos Y values indicate the position of the pivot relative to the anchors.

When the anchors are separated the fields can change partially or completely to Left, Right, Top and Bottom. These fields define the padding inside the rectangle defined by the anchors. The Left and Right fields are used if the anchors are separated horizontally and the Top and Bottom fields are used if they are separated vertically.

Note that changing the values in the anchor or pivot fields will normally counter-adjust the positioning values in order to make the rectangle stay in place. In cases where this is not desired, enable **Raw edit mode** by clicking the **R** button in the Inspector. This causes the anchor and pivot value to be able to be changed without any other

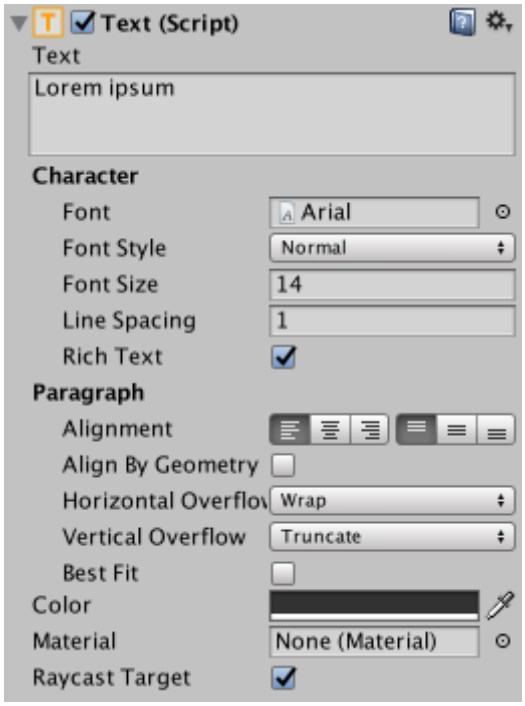
values changing as a result. This will likely cause the rectangle to be visually moved or resized, since its position and size is dependent on the anchor and pivot values.

Visual Components

Leave feedback

With the introduction of the **UI** system, new **Components** have been added that will help you create GUI specific functionality. This section will cover the basics of the new **Components** that can be created.

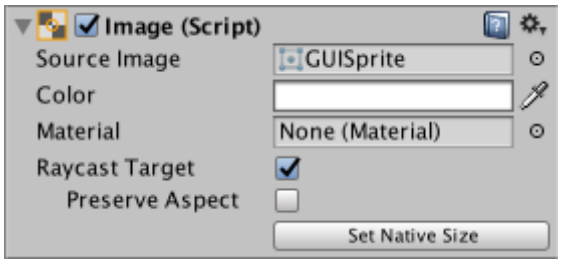
Text



The **Text** component, which is also known as a Label, has a Text area for entering the text that will be displayed. It is possible to set the font, font style, font size and whether or not the text has rich text capability.

There are options to set the alignment of the text, settings for horizontal and vertical overflow which control what happens if the text is larger than the width or height of the rectangle, and a Best Fit option that makes the text resize to fit the available space.

Image



An Image has a Rect **Transform component** and an **Image** component. A **sprite** can be applied to the Image component under the Target Graphic field, and its colour can be set in the Color field. A material can also be applied to the Image component. The Image Type field defines how the applied **sprite** will appear, the options are:

Simple - Scales the whole sprite equally.

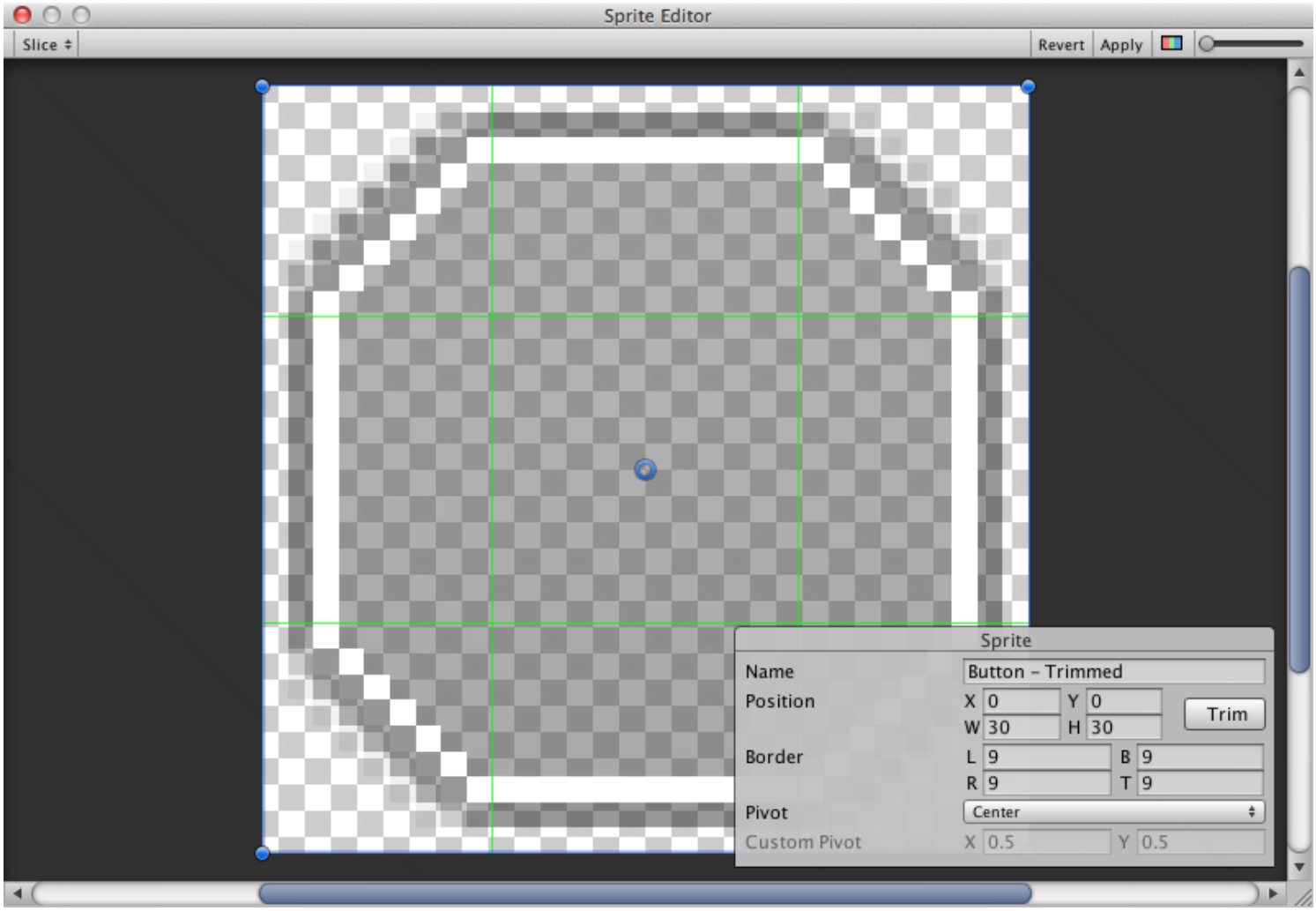
Sliced - Utilises the 3x3 sprite division so that resizing does not distort corners and only the center part is stretched.

Tiled - Similar to Sliced, but tiles (repeats) the center part rather than stretching it. For sprites with no borders at all, the entire sprite is tiled.

Filled - Shows the sprite in the same way as Simple does except that it fills in the sprite from an origin in a defined direction, method and amount.

The option to Set Native Size, which is shown when Simple or Filled is selected, resets the image to the original sprite size.

Images can be imported as **UI sprites** by selecting Sprite(2D / UI) from the 'Texture Type' settings. Sprites have extra import settings compared to the old GUI sprites, the biggest difference is the addition of the sprite editor. The sprite editor provides the option of **9-slicing** the image, this splits the image into 9 areas so that if the sprite is resized the corners are not stretched or distorted.



Raw Image

The Image component takes a sprite but **Raw Image** takes a texture (no borders etc). Raw Image should only be used if necessary otherwise Image will be suitable in the majority of cases.

Mask

A Mask is not a visible UI control but rather a way to modify the appearance of a control's child elements. The mask restricts (ie, "masks") the child elements to the shape of the parent. So, if the child is larger than the parent

then only the part of the child that fits within the parent will be visible.

Effects

Visual components can also have various simple effects applied, such as a simple drop shadow or outline. See the UI Effects reference page for more information.

Interaction Components

Leave feedback

This section covers **components** in the **UI** system that handles interaction, such as mouse or touch events and interaction using a keyboard or controller.

The interaction components are not visible on their own, and must be combined with one or more visual elements in order to work correctly.

Common Functionality

Most of the interaction components have some things in common. They are selectables, which means they have shared built-in functionality for visualising transitions between states (normal, highlighted, pressed, disabled), and for navigation to other selectables using keyboard or controller. This shared functionality is described on the Selectable page.

The interaction components have at least one UnityEvent that is invoked when user interacts with the component in specific way. The UI system catches and logs any exceptions that propagate out of code attached to UnityEvent.

Button

A Button has an **OnClick** UnityEvent to define what it will do when clicked.

See the Button page for details on using the Button component.

Toggle

A Toggle has an **Is On** checkbox that determines whether the Toggle is currently on or off. This value is flipped when the user clicks the Toggle, and a visual checkmark can be turned on or off accordingly. It also has an **OnValueCHanged** UnityEvent to define what it will do when the value is changed.

See the Toggle page for details on using the Toggle component.

Toggle Group

A Toggle Group can be used to group a set of Toggles that are mutually exclusive. Toggles that belong to the same group are constrained so that only one of them can be selected at a time - selecting one of them automatically deselects all the others.

See the Toggle Group page for details on using the Toggle Group component.

Slider

A Slider has a decimal number **Value** that the user can drag between a minimum and maximum value. It can be either horizontal or vertical. It also has a **OnValueChanged** UnityEvent to define what it will do when the value is changed.

See the Slider page for details on using the Slider component.

Scrollbar

A Scrollbar has a decimal number **Value** between 0 and 1. When the user drags the scrollbar, the value changes accordingly.

Scrollbars are often used together with a Scroll Rect and a Mask to create a scroll view. The Scrollbar has a **Size** value between 0 and 1 that determines how big the handle is as a fraction of the entire scrollbar length. This is often controlled from another component to indicate how big a proportion of the content in a scroll view is visible. The Scroll Rect component can automatically do this.

The Scrollbar can be either horizontal or vertical. It also has a **OnValueChanged** UnityEvent to define what it will do when the value is changed.

See the Scrollbar page for details on using the Scrollbar component.

Dropdown

A Dropdown has a list of options to choose from. A text string and optionally an image can be specified for each option, and can be set either in the **Inspector** or dynamically from code. It has a **OnValueChanged** UnityEvent to define what it will do when the currently chosen option is changed.

See the Dropdown page for details on using the Dropdown component.

Input Field

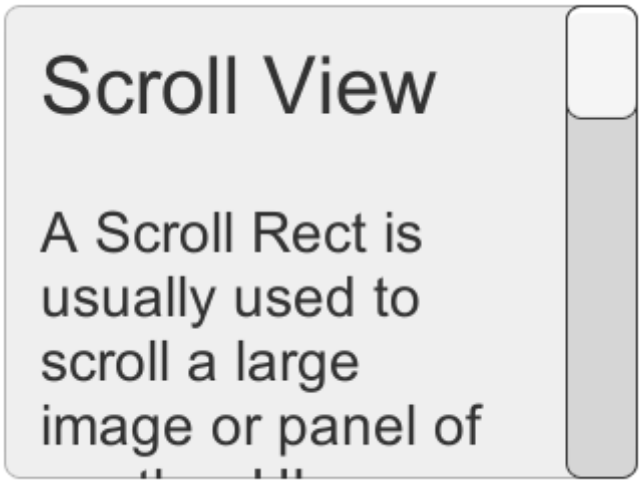
An Input Field is used to make the text of a Text Element editable by the user. It has a UnityEvent to define what it will do when the text content is changed, and an another to define what it will do when the user has finished editing it.

See the Input Field page for details on using the Input Field component.

Scroll Rect (Scroll View)

A Scroll Rect can be used when content that takes up a lot of space needs to be displayed in a small area. The Scroll Rect provides functionality to scroll over this content.

Usually a Scroll Rect is combined with a Mask in order to create a scroll view, where only the scrollable content inside the Scroll Rect is visible. It can also additionally be combined with one or two Scrollbars that can be dragged to scroll horizontally or vertically.

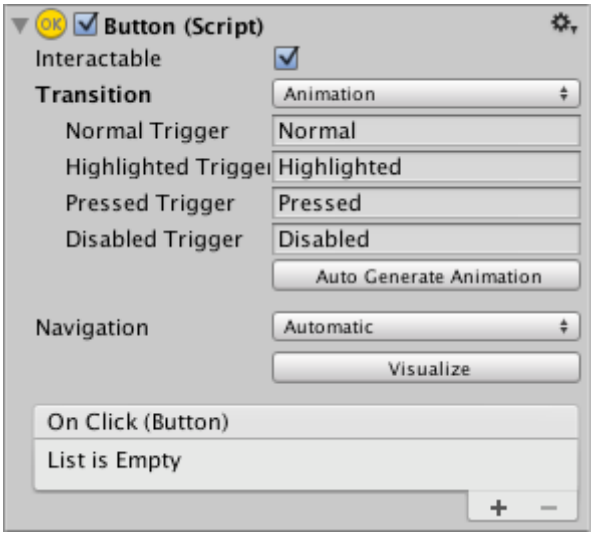


See the Scroll Rect page for details on using the Scroll Rect component.

Animation Integration

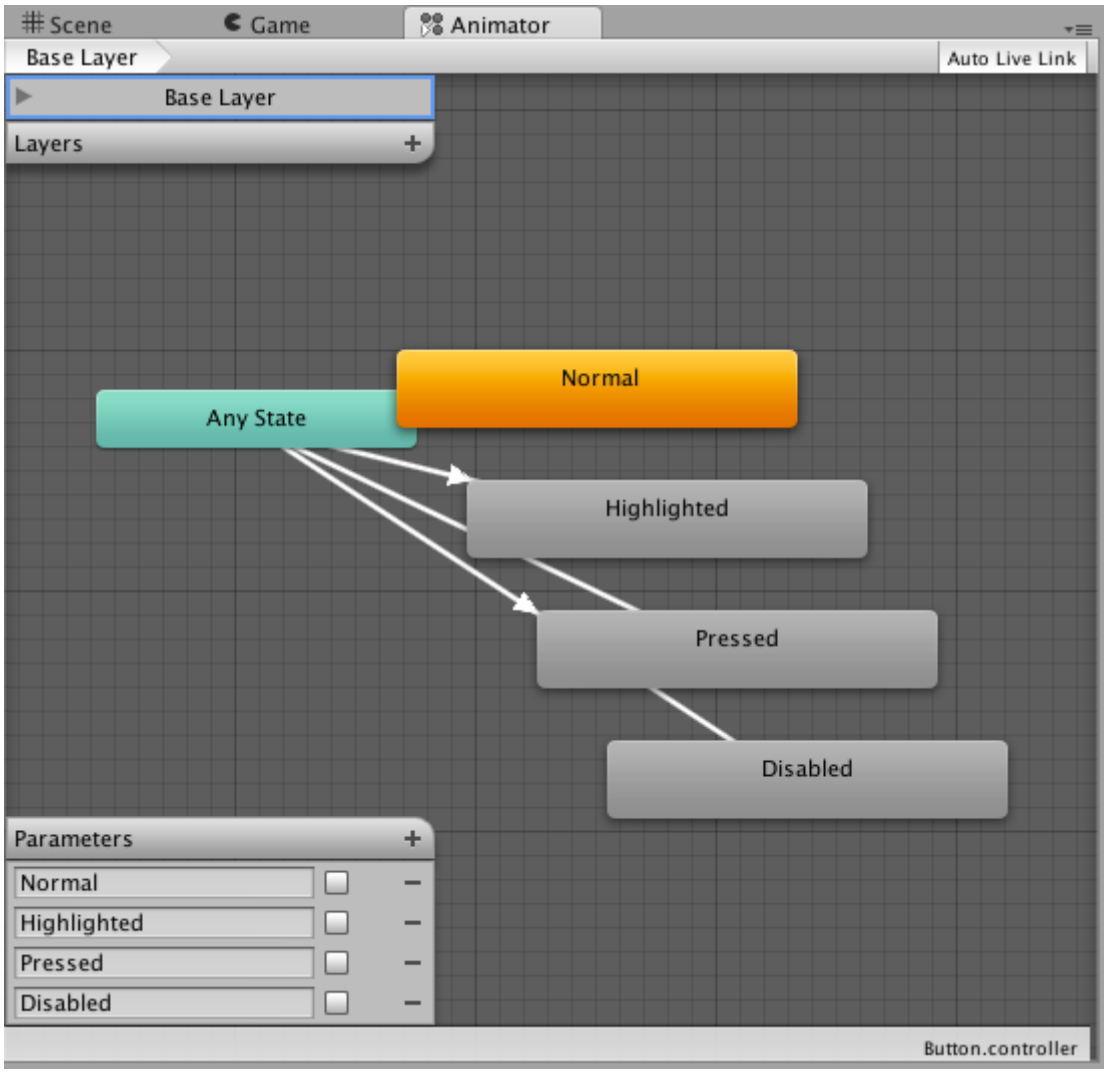
Leave feedback

Animation allows for each transition between control states to be fully animated using Unity's animation system. This is the most powerful of the transition modes due to the number of properties that can be animated simultaneously.



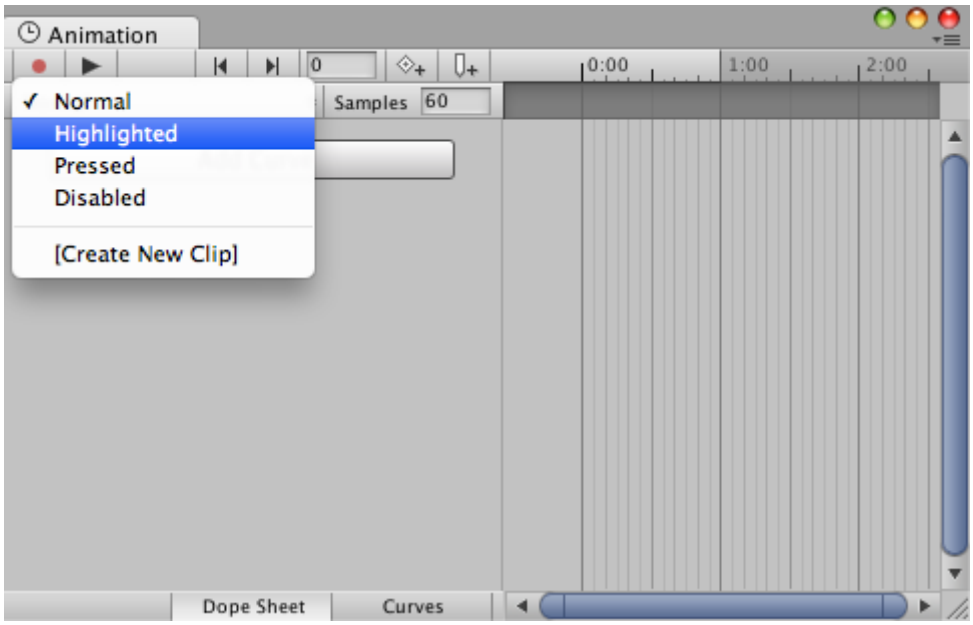
To use the **Animation transition** mode, an Animator **Component** needs to be attached to the controller element. This can be done automatically by clicking "Auto Generate Animation". This also generates an **Animator Controller** with states already set up, which will need to be saved.

The new Animator controller is ready to use straight away. Unlike most Animator Controllers, this controller also stores the animations for the controller's transitions and these can be customised, if desired.



For example, if a Button element with an Animator controller attached is selected, the animations for each of the button's states can be edited by opening the Animation window (**Window>Animation**).

There is an **Animation Clip** pop-up menu to select the desired clip. Choose from "Normal", "Highlighted", "Pressed" and "Disabled".



The Normal State is set by the values on button element itself and can be left empty. On all other states, the most common configuration is a single **keyframe** at the start of the **timeline**. The transition animation between states will be handled by the Animator.

As an example, the width of the button in the Highlighted State could be changed by selecting the Highlighted state from the Animation Clip pop up menu and with the playhead at the start of the time line:

Select the record Button
Change the width of the Button in the inspector
Exit the record mode.
Change to play mode to see how the button grows when highlighted.

Any number of properties can have their parameters set in this one keyframe.

Several buttons can share the same behaviour by sharing Animator Controllers.

The UI Animation transition mode is not compatible with Unity's legacy animation system. You should only use the *Animator* Component.

Auto Layout

Leave feedback

The Rect Transform layout system is flexible enough to handle a lot of different types of layouts and it also allows placing elements in a complete freeform fashion. However, sometimes something a bit more structured can be needed.

The auto layout system provides ways to place elements in nested layout groups such as horizontal groups, vertical groups, or grids. It also allows elements to automatically be sized according to the contained content. For example a button can be dynamically resized to exactly fit its text content plus some padding.

The auto layout system is a system built on top of the basic Rect Transform layout system. It can optionally be used on some or all elements.

Understanding Layout Elements

The auto layout system is based on a concept of **layout elements** and **layout controllers**. A layout element is an Game Object with a Rect Transform and optionally other **components** as well. The layout element has certain knowledge about which size it should have. Layout elements don't directly set their own size, but other **components** that function as layout controllers can use the information they provide in order to calculate a size to use for them.

A layout element has properties that defines its own:

Minimum width
Minimum height
Preferred width
Preferred height
Flexible width
Flexible height

Examples of layout controller components that use the information provided by layout elements are **Content Size Fitter** and the various **Layout Group** components. The basic principles for how layout elements in a layout group are sized is as follows:

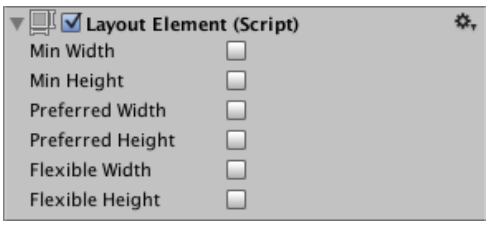
First minimum sizes are allocated.
If there is sufficient available space, preferred sizes are allocated.
If there is additional available space, flexible size is allocated.

Any Game Object with a Rect Transform on it can function as a layout element. They will by default have minimum, preferred, and flexible sizes of 0. Certain components will change these layout properties when added to the Game Object.

The Image and Text components are two examples of components that provide layout element properties. They change the preferred width and height to match the **sprite** or text content.

Layout Element Component

If you want to override the minimum, preferred, or flexible size, you can do that by adding a Layout Element component to the Game Object.



The Layout Element component lets you override the values for one or more of the layout properties. Enable the checkbox for a property you want to override and then specify the value you want to override with.

See the reference page for Layout Element for more information.

Understanding Layout Controllers

Layout controllers are components that control the sizes and possibly positions of one or more layout elements, meaning Game Objects with Rect Transforms on. A layout controller may control its **own layout element** (the same Game Object it is on itself) or it may control **child layout elements**.

A component that functions as a layout controller may also itself function as a layout element at the same time.

Content Size Fitter

The Content Size Fitter functions as a layout controller that controls the size of its own layout element. The simplest way to see the auto layout system in action is to add a Content Size Fitter component to a Game Object with a Text component.

If you set either the Horizontal Fit or Vertical Fit to Preferred, the Rect Transform will adjust its width and/or height to fit the Text content.

See the reference page for Content Size Fitter for more information.

Aspect Ratio Fitter

The **Aspect Ratio** Fitter functions as a layout controller that controls the size of its own layout element.

It can adjust the height to fit the width or vice versa, or it can make the element fit inside its parent or envelope its parent. The Aspect Ratio Fitter does not take layout information into account such as minimum size and preferred size.

See the reference page for Aspect Ratio Fitter for more information.

Layout Groups

A layout group functions as a layout controller that controls the sizes and positions of its child layout elements. For example, a Horizontal Layout Group places its children next to each other, and a Grid Layout Group places its children in a grid.

A layout group doesn't control its own size. Instead it functions as a layout element itself which may be controlled by other layout controllers or be set manually.

Whatever size a layout group is allocated, it will in most cases try to allocate an appropriate amount of space for each of its child layout elements based on the minimum, preferred, and flexible sizes they reported. Layout groups can also be nested arbitrarily this way.

See the reference pages for Horizontal Layout Group, Vertical Layout Group and Grid Layout Group for more information.

Driven Rect Transform properties

Since a layout controller in the auto layout system can automatically control the sizes and placement of certain **UI** elements, those sizes and positions should not be manually edited at the same time through the **Inspector** or **Scene View**. Such changed values would just get reset by the layout controller on the next layout calculation anyway.

The Rect Transform has a concept of **driven properties** to address this. For example, a Content Size Fitter which has the Horizontal Fit property set to Minimum or Preferred will drive the width of the Rect Transform on the same Game Object. The width will appear as read-only and a small info box at the top of the Rect Transform will inform that one or more properties are driven by Conten Size Fitter.

The driven Rect Transforms properties have other reasons beside preventing manual editing. A layout can be changed just by changing the resolution or size of the Game View. This in turn can change the size or placement of layout elements, which changes the values of driven properties. But it wouldn't be desirable that the **Scene** is marked as having unsaved changes just because the Game View was resized. To prevent this, the values of driven properties are not saved as part of the **Scene** and changes to them do not mark the **scene** as changed.

Technical Details

The auto layout system comes with certain components built-in, but it is also possible to create new components that controls layouts in custom ways. This is done by having a component implement specific interfaces which are recognized by the auto layout system.

Layout Interfaces

A component is treated as a layout element by the auto layout system if it implements the interface **ILayoutElement**.

A component is expected to drive the Rect Transforms of its children if it implements the interface **ILayoutGroup**.

A component is expected to drive its own RectTransform if it implements the interface **ILayoutSelfController**.

Layout Calculations

The auto layout system evaluates and executes layouts in the following order:

The minimum, preferred, and flexible widths of layout elements are calculated by calling CalculateLayoutInputHorizontal on ILayoutElement components. This is performed in bottom-up order, where children are calculated before their parents, such that the parents may take the information in their children into account in their own calculations.
The effective widths of layout elements are calculated and set by calling SetLayoutHorizontal on ILayoutController components. This is performed in top-down order, where children are calculated after their parents, since allocation of child widths needs to be based on the full width available in the parent. After this step the Rect Transforms of the layout elements have their new widths.
The minimum, preferred, and flexible heights of layout elements are calculated by calling CalculateLayoutInputVertical on ILayoutElement components. This is performed in bottom-up order, where children are calculated before their parents, such that the parents may take the information in their children into account in their own calculations.
The effective heights of layout elements are calculated and set by calling SetLayoutVertical on ILayoutController components. This is performed in top-down order, where children are calculated after their parents, since allocation of child heights needs to be based on the full height available in the parent. After this step the Rect Transforms of the layout elements have their new heights.
As can be seen from the above, the auto layout system evaluates widths first and then evaluates heights afterwards. Thus, calculated heights may depend on widths, but calculated widths can never depend on heights.

Triggering Layout Rebuild

When a property on a component changes which can cause the current layout to no longer be valid, a layout recalculation is needed. This can be triggered using the call:

LayoutRebuilder.MarkLayoutForRebuild (transform as RectTransform);

The rebuild will not happen immediately, but at the end of the current frame, just before **rendering** happens. The reason it is not immediate is that this would cause layouts to be potentially rebuild many times during the same frame, which would be bad for performance.

Guidelines for when a rebuild should be triggered:

In setters for properties that can change the layout.
In these callbacks:
OnEnable
OnDisable
OnRectTransformDimensionsChange
OnValidate (only needed in the editor, not at runtime)
OnDidApplyAnimationProperties

Rich Text

Leave feedback

The text for **UI** elements and text meshes can incorporate multiple font styles and sizes. Rich text is supported both for the **UI** System and the legacy GUI system. The Text, GUIStyle, GUIText and TextMesh classes have a **Rich Text** setting which instructs Unity to look for markup tags within the text. The Debug.Log function can also use these markup tags to enhance error reports from code. The tags are not displayed but indicate style changes to be applied to the text.

Markup format

The markup system is inspired by HTML but isn't intended to be strictly compatible with standard HTML. The basic idea is that a section of text can be enclosed inside a pair of matching tags:-

```
We are <b>not</b> amused
```

As the example shows, the tags are just pieces of text inside the "angle bracket" characters, < and >. The text inside the tag denotes its name (which in this case is just **b**). Note that the tag at the end of the section has the same name as the one at the start but with the slash / character added. The tags are not displayed to the user directly but are interpreted as instructions for styling the text they enclose. The b tag used in the example above applies boldface to the word "not", so the text will appear onscreen as:-

We are **not** amused

A marked up section of text (including the tags that enclose it) is referred to as an **element**.

Nested elements

It is possible to apply more than one style to a section of text by "nesting" one element inside another

```
We are <b><i>definitely not</i></b> amused
```

The i tag applies italic style, so this would be presented onscreen as

We are ***definitely not*** amused

Note the ordering of the ending tags, which is in reverse to that of the starting tags. The reason for this is perhaps clearer when you consider that the inner tags need not span the whole text of the outermost element

```
We are <b>absolutely <i>definitely</i> not</b> amused
```

which gives

We are **absolutely *definitely* not** amused

Tag parameters

Some tags have a simple all-or-nothing effect on the text but others might allow for variations. For example, the **color** tag needs to know which color to apply. Information like this is added to tags by the use of **parameters**:-

```
We are <color=green>green</color> with envy
```

Note that the ending tag doesn't include the parameter value. Optionally, the value can be surrounded by quotation marks but this isn't required.

Supported tags

The following list describes all the styling tags supported by Unity.

Tag	*Description*	*Example*	*Notes*
b	Renders the text in boldface.	We are not amused.	
i	Renders the text in italics.	We are usually not amused.	
size	Sets the size of the text according to the parameter value, given in pixels.	We are <size=50>largely</size> unaffected.	Although this tag is available for Debug.Log, you will find that the line spacing in the window bar and Console looks strange if the size is set too large.
color	Sets the color of the text according to the parameter value. The color can be specified in the traditional HTML format. *#rrggbbaa* ...where the letters correspond to pairs of hexadecimal digits denoting the red, green, blue and alpha (transparency) values for the color. For example, cyan at full opacity would be specified by	*<color=#00ffffff>...*	Another option is to use the name of the color. This is easier to understand but naturally, the range of colors is limited and full opacity is always assumed. *<color=cyan>...* The available color names are given in the table below.

Color name	*Hex value*	*Swatch*
aqua (same as cyan)	`#00ffffff`	
black	`#000000ff`	
blue	`#0000ffff`	
brown	`#a52a2aff`	
cyan (same as aqua)	`#00ffffff`	

Color name	*Hex value*	*Swatch*
darkblue	`#0000a0ff`	
fuchsia (same as magenta)	`#ff00ffff`	
green	`#008000ff`	
grey	`#808080ff`	
lightblue	`#add8e6ff`	
lime	`#00ff00ff`	
magenta (same as fuchsia)	`#ff00ffff`	
maroon	`#800000ff`	
navy	`#000080ff`	
olive	`#808000ff`	
orange	`#ffa500ff`	
purple	`#800080ff`	
red	`#ff0000ff`	
silver	`#c0c0c0ff`	
teal	`#008080ff`	
white	`#ffffffff`	
yellow	`#ffff00ff`	

material

This is only useful for text meshes and renders a section of text with a material specified by the parameter. The value is an index into the text **mesh**'s array of materials as shown by the **inspector**.

```
We are <material=2>texturally</material> amused
```

quad

This is only useful for text meshes and renders an image inline with the text. It takes parameters that specify the material to use for the image, the image height in **pixels**, and a further four that denote a rectangular area of the image to display. Unlike the other tags, quad does not surround a piece of text and so there is no ending tag - the slash character is placed at the end of the initial tag to indicate that it is "self-closing".

```
<quad material=1 size=20 x=0.1 y=0.1 width=0.5 height=0.5 />
```

This selects the material at position in the renderer's material array and sets the height of the image to 20 pixels. The rectangular area of image starts at given by the x, y, width and height values, which are all given as a fraction of the unscaled width and height of the texture.

Editor GUI

Rich text is disabled by default in the editor GUI system but it can be enabled explicitly using a custom GUIStyle. The richText property should be set to true and the style passed to the GUI function in question:-

```
GUIStyle style = new GUIStyle ();
style.richText = true;
GUILayout.Label("<size=30>Some <color=yellow>RICH</color> text</size>",style);
```

UI Reference

Leave feedback

This section goes into more depth about Unity's **UI** features.

Rect Transform

Leave feedback

SWITCH TO SCRIPTING

The **Rect Transform component** is the 2D layout counterpart of the Transform component. Where Transform represents a single point, Rect Transform represent a rectangle that a **UI** element can be placed inside. If the parent of a Rect Transform is also a Rect Transform, the child Rect Transform can also specify how it should be positioned and sized relative to the parent rectangle.

Properties

Property:	*Function:*
Pos (X, Y, Z)	Position of the rectangle's pivot point relative to the anchors.
Width/Height	Width and height of the rectangle.
Left, Top, Right, Bottom	Positions of the rectangle's edges relative to their anchors. This can be thought of as padding inside the rectangle defined by the anchors. Shown in place of *Pos* and *Width/Height* when the anchors are separated (see below).
Anchors	The anchor points for the lower left corner and the upper right corner of the rectangle.
Min	The anchor point for the lower left corner of the rectangle defined as a fraction of the size of the parent rectangle. 0,0 corresponds to anchoring to the lower left corner of the parent, while 1,1 corresponds to anchoring to the upper right corner of the parent.
Max	The anchor point for the upper right corner of the rectangle defined as a fraction of the size of the parent rectangle. 0,0 corresponds to anchoring to the lower left corner of the parent, while 1,1 corresponds to anchoring to the upper right corner of the parent.
Pivot	Location of the pivot point around which the rectangle rotates, defined as a fraction of the size of the rectangle itself. 0,0 corresponds to the lower left corner while 1,1 corresponds to the upper right corner.
Rotation	Angle of rotation (in degrees) of the object around its pivot point along the X, Y and Z axis.
Scale	Scale factor applied to the object in the X, Y and Z dimensions.

Details

Note that some RectTransform calculations are performed at the end of a frame, just before calculating UI vertices, in order to ensure that they are up to date with all the latest changes performed throughout the frame. This means that they haven't yet been calculated for the first time in the Start callback and first Update callback.

You can work around this by creating a `Start()` callback and adding `Canvas.ForceUpdateCanvases()` method to it. This will force Canvas to be updated not at the end of the frame, but when that method is called.

See the Basic Layout page for a full introduction and overview of how to use the Rect Transform.

Canvas Components

Leave feedback

All **UI Components** are placed within a Canvas.

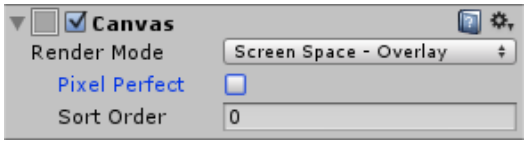
Canvas
Canvas Scaler
Canvas Group
Canvas Renderer

Canvas

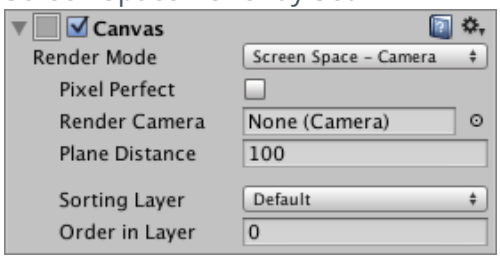
Leave feedback

SWITCH TO SCRIPTING

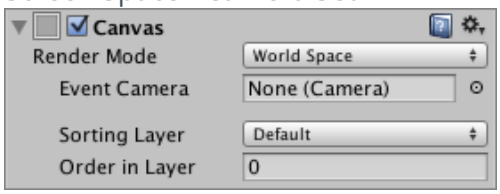
The **Canvas** component represents the abstract space in which the UI is laid out and rendered. All UI elements must be children of a GameObject that has a Canvas component attached. When you create a UI element object from the menu (**GameObject > Create UI**), a Canvas object will be created automatically if there isn't one in the scene already.



Screen Space - Overlay Set



Screen Space - Camera Set



World Space Set

Properties

Property:	*Function:*
Render Mode	The way the **UI** is rendered to the screen or as an object in 3D space (see below). The options are *Screen Space - Overlay*, *Screen Space - Camera* and *World Space*.
Pixel Perfect (*Screen Space* modes only)	Should the UI be rendered without **antialiasing** for precision?
Render Camera__ (*Screen Space - Camera* mode only)__	The camera to which the UI should be rendered (see below).
Plane Distance (*Screen Space - Camera* mode only)	The distance at which the UI plane should be placed in front of the camera.
Event Camera (*World Space* mode only)	The camera that will be used to process UI events.
Receives Events	Are UI events processed by this Canvas?

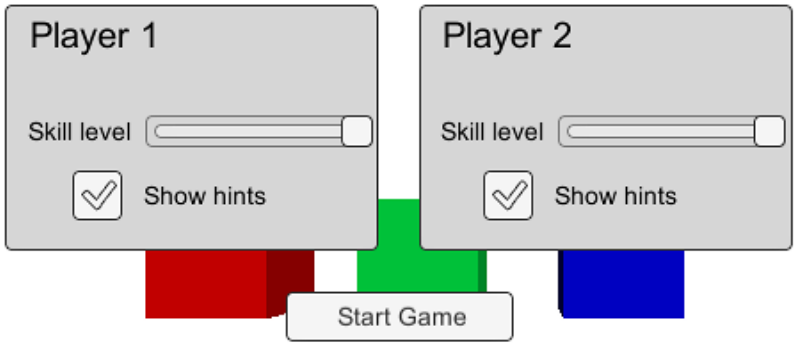
Details

A single Canvas for all UI elements is sufficient but multiple Canvases in the **scene** is possible. It is also possible use nested Canvases, where one Canvas is placed as a child of another for optimization purposes. A nested Canvas uses the same Render Mode as its parent.

Traditionally, UIs are rendered as if they were simple graphic designs drawn directly on the screen. That is to say, they have no concept of a 3D space being viewed by a camera. Unity supports this kind of screen space **rendering** but also allows UIs to rendered as objects in the scene, depending on the value of the *Render Mode* property. The modes available are **Screen Space - Overlay**, **Screen Space - Camera** and **World Space**.

Screen Space - Overlay

In this mode, the Canvas is scaled to fit the screen and then rendered directly without reference to the scene or a camera (the UI will be rendered even if there is no camera in the scene at all). If the screen's size or resolution are changed then the UI will automatically rescale to fit. The UI will be drawn over any other graphics such as the camera view.

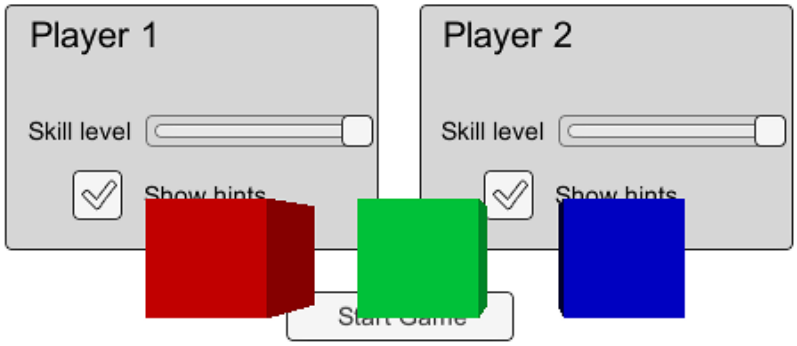


Overlay UI rendered over scene objects

Note: The Screen Space - Overlay canvas needs to be stored at the top level of the hierarchy. If this is not used then the UI may disappear from the view. This is a built-in limitation. Keep the Screen Space - Overlay canvas at the top level of the hierarchy to get expected results.

Screen Space - Camera

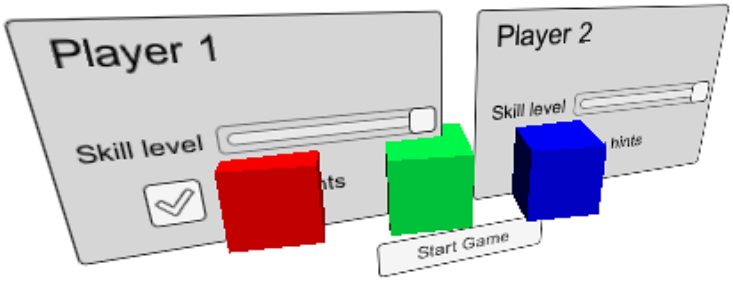
In this mode, the Canvas is rendered as if it were drawn on a plane object some distance in front of a given camera. The onscreen size of the UI does not vary with the distance since it is always rescaled to fit exactly within the camera frustum. If the screen's size or resolution or the camera frustum are changed then the UI will automatically rescale to fit. Any **3D objects** in the scene that are closer to the camera than the UI plane will be rendered in front of the UI, while objects behind the plane will be obscured.



Camera mode UI with scene objects in front

World Space

This mode renders the UI as if it were a plane object in the scene. Unlike *Screen Space - Camera* mode, however, the plane need not face the camera and can be oriented however you like. The size of the Canvas can be set using its Rect Transform but its onscreen size will depend on the viewing angle and distance of the camera. Other scene objects can pass behind, through or in front of the Canvas.



World space UI intersecting scene objects

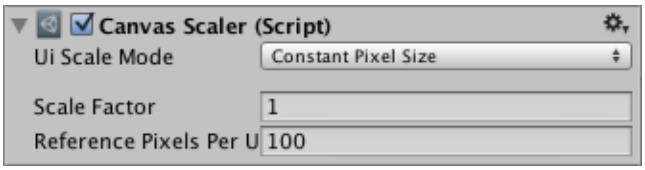
Hints

Read more about setting up a World Space Canvas on the Creating a World Space UI page.
For information about making your Canvas and UI scale to different resolutions or **aspect ratios**, see the Designing UI for Multiple Resolutions page as well as the Canvas Scaler page.

Canvas Scaler

Leave feedback

The **Canvas Scaler component** is used for controlling the overall scale and **pixel** density of **UI** elements in the Canvas. This scaling affects everything under the Canvas, including font sizes and image borders.



Properties

Property:	*Function:*
UI Scale Mode	Determines how UI elements in the Canvas are scaled.
Constant Pixel Size	Makes UI elements retain the same size in pixels regardless of screen size.
Scale With Screen Size	Makes UI elements bigger the bigger the screen is.
Constant Physical Size	Makes UI elements retain the same physical size regardless of screen size and resolution.

Settings for Constant Pixel Size:

Property:	*Function:*
Scale Factor	Scales all UI elements in the Canvas by this factor.
Reference Pixels Per Unit	If a **sprite** has this 'Pixels Per Unit' setting, then one pixel in the **sprite** will cover one unit in the UI.

Settings for Scale With Screen Size:

Property:	*Function:*
Reference Resolution	The resolution the UI layout is designed for. If the screen resolution is larger, the UI will be scaled up, and if it's smaller, the UI will be scaled down.
Screen Match Mode	A mode used to scale the canvas area if the **aspect ratio** of the current resolution doesn't fit the reference resolution.
Match Width or Height	Scale the canvas area with the width as reference, the height as reference, or something in between.
Expand	Expand the canvas area either horizontally or vertically, so the size of the canvas will never be smaller than the reference.
Shrink	Crop the canvas area either horizontally or vertically, so the size of the canvas will never be larger than the reference.
Match	Determines if the scaling is using the width or height as reference, or a mix in between.
Reference Pixels Per Unit	If a sprite has this 'Pixels Per Unit' setting, then one pixel in the sprite will cover one unit in the UI.

Settings for Constant Physical Size:

Property:	*Function:*
Physical Unit	The physical unit to specify positions and sizes in.
Fallback Screen DPI	The DPI to assume if the screen DPI is not known.

Property:	*Function:*
Default Sprite DPI	The pixels per inch to use for sprites that have a 'Pixels Per Unit' setting that matches the 'Reference Pixels Per Unit' setting.
Reference Pixels Per Unit	If a sprite has this 'Pixels Per Unit' setting, then its DPI will match the 'Default Sprite DPI' setting.

Settings for World Space Canvas (shown when Canvas component is set to World Space):

Property:	*Function:*
Dynamic Pixels Per Unit	The amount of pixels per unit to use for dynamically created bitmaps in the UI, such as Text.
Reference Pixels Per Unit	If a sprite has this 'Pixels Per Unit' setting, then one pixel in the sprite will cover one unit in the world. If the 'Reference Pixels Per Unit' is set to 1, then the 'Pixels Per Unit' setting in the sprite will be used as-is.

Details

For a Canvas set to 'Screen Space - Overlay' or 'Screen Space - **Camera**', the Canvas Scaler UI Scale Mode can be set to Constant Pixel Size, Scale With Screen Size, or Constant Physical Size.

Constant Pixel Size

Using the Constant Pixel Size mode, positions and sizes of UI elements are specified in pixels on the screen. This is also the default functionality of the Canvas when no Canvas Scaler is attached. However, With the Scale Factor setting in the Canvas Scaler, a constant scaling can be applied to all UI elements in the Canvas.

Scale With Screen Size

Using the Scale With Screen Size mode, positions and sizes can be specified according to the pixels of a specified reference resolution. If the current screen resolution is larger than the reference resolution, the Canvas will keep having only the resolution of the reference resolution, but will scale up in order to fit the screen. If the current screen resolution is smaller than the reference resolution, the Canvas will similarly be scaled down to fit.

If the current screen resolution has a different aspect ratio than the reference resolution, scaling each axis individually to fit the screen would result in non-uniform scaling, which is generally undesirable. Instead of this, the ReferenceResolution component will make the Canvas resolution deviate from the reference resolution in order to respect the aspect ratio of the screen. It is possible to control how this deviation should behave using the Screen Match Mode setting.

Constant Physical Size

Using the Constant Physical Size mode, positions and sizes of UI elements are specified in physical units, such as millimeters, points, or picas. This mode relies on the device reporting its screen DPI correctly. You can specify a fallback DPI to use for devices that do not report a DPI.

World Space

For a Canvas set to 'World Space' the Canvas Scaler can be used to control the pixel density of UI elements in the Canvas.

Hints

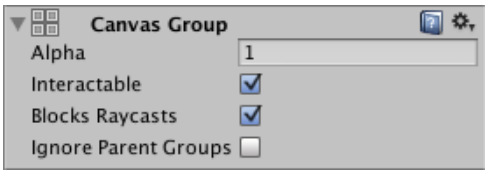
See the page Designing UI for Multiple Resolutions for a step by step explanation of how Rect Transform anchoring and Canvas Scaler can be used in conjunction to make UI layouts that adapt to different resolutions and aspect ratios.

Canvas Group

Leave feedback

SWITCH TO SCRIPTING

The **Canvas Group** can be used to control certain aspects of a whole group of **UI** elements from one place without needing to handle them each individually. The properties of the **Canvas Group** affect the **GameObject** it is on as well as all children.



Properties

Property:	*Function:*
Alpha	The opacity of the UI elements in this group. The value is between 0 and 1 where 0 is fully transparent and 1 is fully opaque. Note that elements retain their own transparency as well, so the Canvas Group alpha and the alpha values of the individual UI elements are multiplied with each other.
Interactable	Determines if this **component** will accept input. When it is set to false interaction is disabled.
Block Raycasts	Will this component act as a **collider** for Raycasts? You will need to call the RayCast function on the graphic raycaster attached to the Canvas. This does *not* apply to **Physics.Raycast**.
Ignore Parent Groups	Will this group also be affected by the settings in Canvas Group components further up in the Game Object hierarchy, or will it ignore those and hence override them?

Details

Typical uses of Canvas Group are:

Fading in or out a whole window by adding a Canvas Group on the GameObject of the Window and control its Alpha property.
Making a whole set of controls non-interactable (“grayed out”) by adding a Canvas Group to a parent GameObject and setting its Interactable property to false.
Making one or more UI elements not block mouse events by placing a Canvas Group component on the element or one of its parents and setting its Block Raycasts property to false.

Canvas Renderer

Leave feedback

SWITCH TO SCRIPTING

The **Canvas Renderer component** renders a graphical **UI** object contained within a Canvas.

Properties

The Canvas Renderer has no properties exposed in the **inspector**.

Details

The standard UI objects available from the menu (**GameObject > Create UI**) all have Canvas Renderers attached wherever they are required but you may need to add this component manually for custom UI objects. Although there are no properties exposed in the inspector, a few properties and function can be accessed from scripts - see the CanvasRenderer page in the Script Reference for full details.

Visual Components

Leave feedback

The visual components allow for ease of creation and GUI specific functionality.

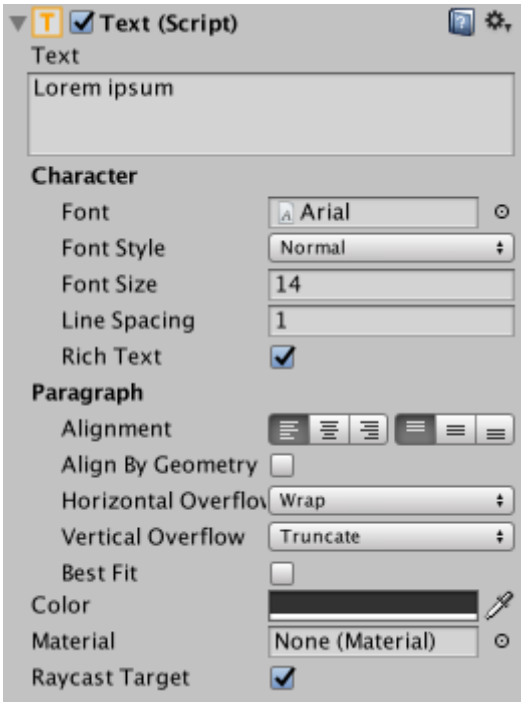
Text
Image
Raw Image
Mask

Text

Leave feedback

The **Text** control displays a non-interactive piece of text to the user. This can be used to provide captions or labels for other GUI controls or to display instructions or other text.

Properties



Property:	*Function:*
Text	The text displayed by the control.
Character	
Font	The Font used to display the text.
Font Style	The style applied to the text. The options are *Normal*, *Bold*, *Italic* and *Bold And Italic*.
Font Size	The size of the displayed text.
Line Spacing	The vertical separation between lines of text.
Rich Text	Should markup elements in the text be interpreted as Rich Text styling?
Paragraph	
Alignment	The horizontal and vertical alignment of the text.
Align by Geometry	Use the extents of glyph geometry to perform horizontal alignment rather than glyph metrics.
Horizontal Overflow	The method used to handle the situation where the text is too wide to fit in the rectangle. The options are *Wrap* and *Overflow*.
Vertical Overflow	The method used to handle the situation where wrapped text is too tall to fit in the rectangle. The options are *Truncate* and *Overflow*.
Best Fit	Should Unity ignore the size properties and simply try to fit the text to the control's rectangle?
Color	The color used to render the text.
Material	The Material used to render the text.

A default text element looks like this:

Lorem ipsum

A Text element.

Details

Some controls (such as Buttons and Toggles) have textual descriptions built-in. For controls that have no implicit text (such as Sliders), you can indicate the purpose using a label created with a Text control. Text is also useful for lists of instructions, story text, conversations and legal disclaimers.

The Text control offers the usual parameters for font size, style, **etc**, and text alignment. When the *Rich Text* option is enabled, markup elements within the text will be treated as styling information, so you can have just a single word or short section in boldface or in a different color, say (see the page about Rich Text for details of the markup scheme).

Hints

See the Effects page for how to apply a simple shadow or outline effect to the text.

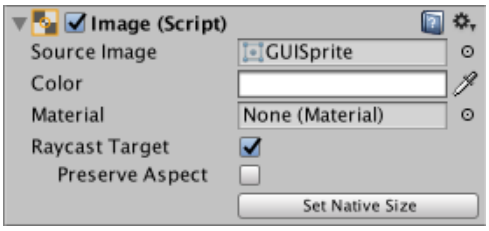
Image

Leave feedback

The **Image** control displays a non-interactive image to the user. This can be used for decoration, icons, **etc**, and the image can also be changed from a script to reflect changes in other controls. The control is similar to the Raw Image control but offers more options for animating the image and accurately filing the control rectangle. However, the Image control requires its texture to be a Sprite, while the Raw Image can accept any texture.

An Image control

Properties



Property:	*Function:*
Source Image	The texture that represents the image to display (which must be imported as a Sprite).
Color	The color to apply to the image.
Material	The Material to use for **rendering** the image.
Raycast Target	Should this be considered a target for raycasting?
Preserve Aspect	Ensure image remains existing dimension.
Set Native Size	Button to set the dimensions of the image box to the original **pixel** size of the texture.

Details

The image to display must be imported as a Sprite to work with the Image control.

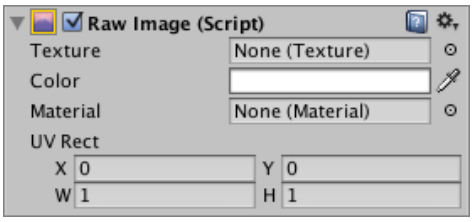
Raw Image

Leave feedback

The **Raw Image** control displays a non-interactive image to the user. This can be used for decoration, icons, **etc**, and the image can also be changed from a script to reflect changes in other controls. The control is similar to the Image control but does not have the same set of options for animating the image and accurately filing the control rectangle. However, the Raw Image can display any texture whilst the Image can only show a Sprite texture.

A Raw Image control

Properties



Property:	*Function:*
Texture	The texture that represents the image to display.
Color	The color to apply to the image.
Material	The Material to use for **rendering** the image.
UV Rectangle	The image's offset and size within the control rectangle, given in normalized coordinates (range 0.0 to 1.0). The edges of the image are stretched to fill the space around the UV rectangle.

Details

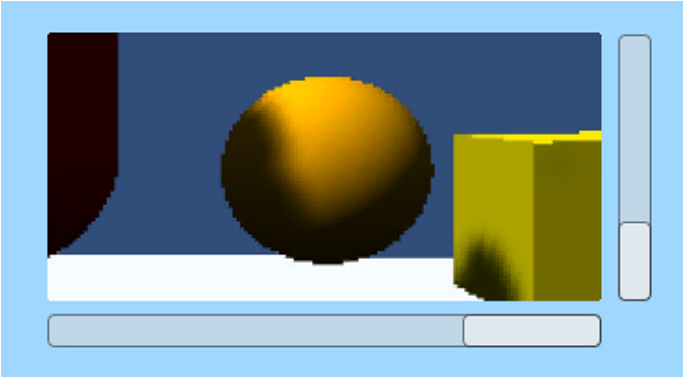
Since the Raw Image does not require a sprite texture, you can use it to display any texture available to the Unity player. For example, you might show an image downloaded from a URL using the WWW class or a texture from an object in a game.

The *UV Rectangle* properties allow you to display a small section of a larger image. The *X* and *Y* coordinates specify which part of the image is aligned with the bottom left corner of the control. For example, an X coordinate of 0.25 will cut off the leftmost quarter of the image. The *W* and *H* (ie, width and height) properties indicate the width and height of the section of image that will be scaled to fit the control rectangle. For example, a width and height of 0.5 will scale a quarter of the image area up to the control rectangle. By changing these properties, you can zoom and scale the image as desired (see also the Scrollbar control).

Mask

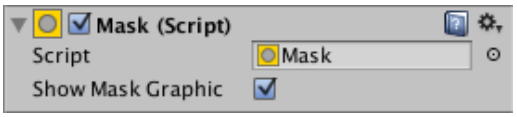
Leave feedback

A **Mask** is not a visible **UI** control but rather a way to modify the appearance of a control's child elements. The mask restricts (ie, "masks") the child elements to the shape of the parent. So, if the child is larger than the parent then only the part of the child that fits within the parent will be visible.



Section of a large Image masked by a Panel (Scrollbars are separate controls)

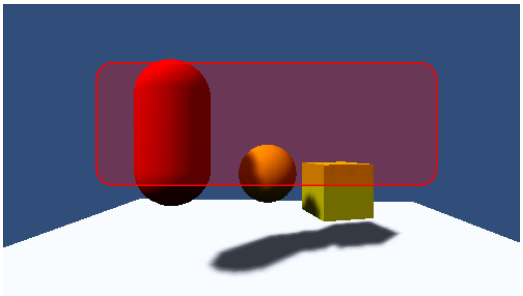
Properties



Property:	*Function:*
Show Graphic	Should the graphic of the masking (parent) object be drawn with alpha over the child object?

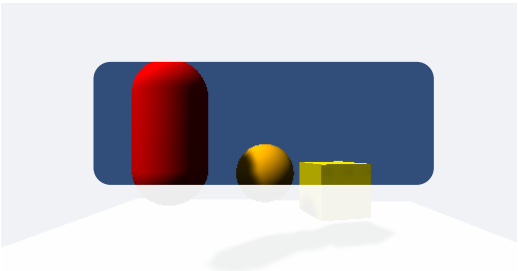
Description

A common use of a Mask is to show a small section of a large Image, using say a Panel object (menu: **GameObject > Create UI > Panel**) as a "frame". You can achieve this by firstly making the Image a child of the Panel object. You should position the Image so that the area that should be visible is directly behind the Panel area.



Panel area shown in red with child Image behind

Then, add a Mask **component** to the Panel. The areas of the child Image outside the panel will become invisible since they are masked by the shape of the Panel.



Masked areas shown faint, but would really be invisible

If the image is then moved around then only the part revealed by the Panel will be visible. The movement could be controlled by Scrollbars to create a scrollable viewer for a map, say.

Implementation

Masking is implemented using the stencil buffer of the GPU.

The first Mask element writes a 1 to the stencil buffer All elements below the mask check when **rendering**, and only render to areas where there is a 1 in the stencil buffer *Nested Masks will write incremental bit masks into the buffer, this means that renderable children need to have the logical & of the stencil values to be rendered.

RectMask2D

Leave feedback

A **RectMask2D** is a masking control similar to the **Mask** control. The mask restricts the child elements to the rectangle of the parent element. Unlike the standard Mask control it has some limitations, but it also has a number of performance benefits.

Description

A common use of a RectMask2D is to show small sections of a larger area. Using the RectMask2D to frame this area.

The limitations of RectMask2D control are:

It only works in 2D space
It will not properly mask elements that are not coplanar
The advantages of RectMask2D are:

It does not use the stencil buffer
No extra draw calls
No material changes
Fast performance

UI Effect Components

Leave feedback

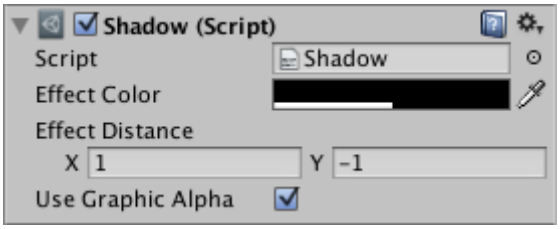
The effects **components** allow adding simple effects to Text and Image graphics, such as shadow and outline.

Shadow
Outline
Position as UV1

Shadow

Leave feedback

The Shadow **component** adds a simple outline effect to graphic components such as Text or Image. It must be on the same **GameObject** as the graphic **component**.



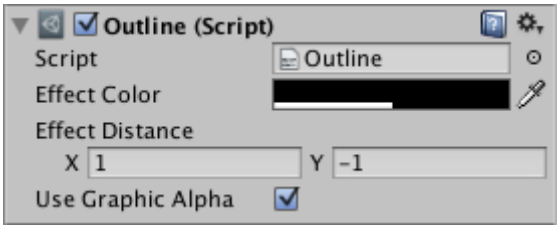
Properties

Property:	*Function:*
Effect Color	The color of the shadow.
Effect Distance	The offset of the shadow expressed as a vector.
Use Graphic Alpha	Multiplies the color of the graphic onto the color of the effect.

Outline

Leave feedback

The Outline **component** adds a simple outline effect to graphic components such as Text or Image. It must be on the same **GameObject** as the graphic **component**.



Properties

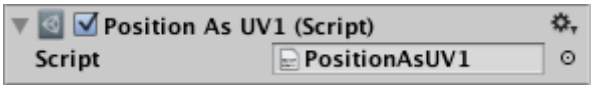
Property:	*Function:*
Effect Color	The color of the outline.
Effect Distance	The distance of the outline effect horizontally and vertically.
Use Graphic Alpha	Multiplies the color of the graphic onto the color of the effect.

Position as UV1

Leave feedback

This adds a simple Position as UV1 effect to text and image graphics.

Properties



Property: Function:

Script

Interaction Components

Leave feedback

The interaction components in the **UI** system handle interaction, such as mouse or touch events and interaction using a keyboard or controller.

Selectable Base Class
Button
Toggle
Toggle Group
Slider
Scrollbar
Scroll Rect
InputField

Selectable Base Class

Leave feedback

The Selectable Class is the base class for all the interaction **components** and it handles the items that are in common.

Property:	*Function:*
Interactible	This determines if this component will accept input. When it is set to false interaction is disabled and the transition state will be set to the disabled state.
Transition	Within a selectable component there are several Transition Options depending on what state the selectable is currently in. The different states are: normal, highlighted, pressed and disabled.
Navigation	There are also a number of Navigation Options to control how keyboard navigation of the controls is implemented.

Transition Options

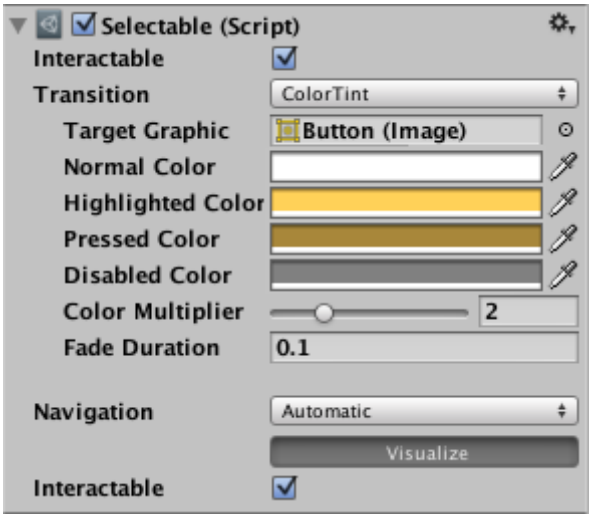
Leave feedback

Within a selectable **component** there are several transition options depending on what state the selectable is currently in. The different states are: normal, highlighted, pressed and disabled.

Transition Options:	*Function:*
None	This option is for the button to have no state effects at all.
Color Tint	Changes the colour of the button depending on what state it is in. It is possible to select the colour for each individual state. It is also possible to set the Fade Duration between the different states. The higher the number is, the slower the fade between colors will be.
Sprite Swap	Allows different **sprites** to display depending on what state the button is currently in, the **sprites** can be customised.
Animation	Allows animations to occur depending on the state of the button, an **animator component** must exist in order to use **animation transition**. It's important to make sure **root motion** is disabled. To create an animation controller click on generate animation (or create your own) and make sure that an animation controller has been added to the animator component of the button.

Each Transition option (except None) provides additional options for controlling the transitions. We'll go into details with those in each of the sections below.

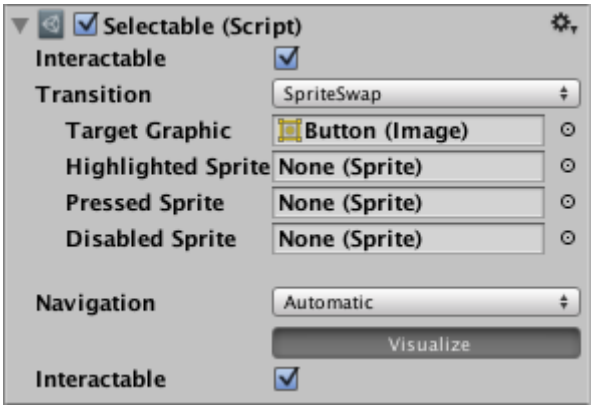
Color Tint



Property:	*Function:*
Target Graphic	The graphic used for the interaction component.
Normal Color	The normal color of the control

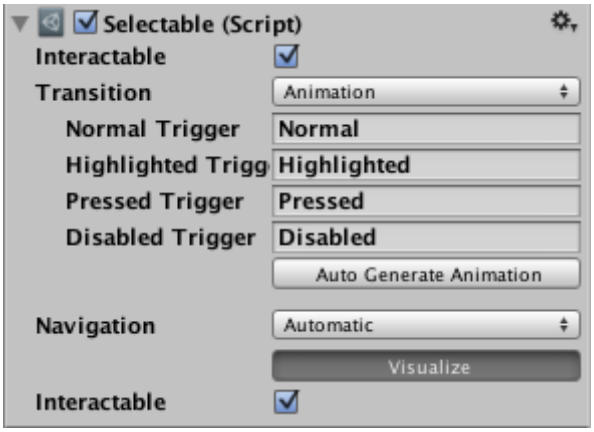
Property:	*Function:*
Highlighted Color	The color of the control when it is highlighted
Pressed Color	The color of the control when it is pressed
Disabled Color	The color of the control when it is disabled
Color Multiplier	This multiplies the tint color for each transition by its value. With this you can create colors greater than 1 to brighten the colors (or alpha channel) on graphic elements whose base color is less than white (or less then full alpha).
Fade Duration	The time taken, in seconds, to fade from one state to another

Sprite Swap



Property:	*Function:*
Target Graphic	The normal sprite to use
Highlighted Sprite	Sprite to use when the control is highlighted
Pressed Sprite	Sprite to use when the control is pressed
Disabled Sprite	Sprite to use when the control is disabled

Animation

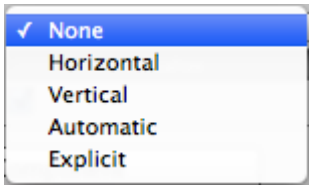


Property:	*Function:*
Normal Trigger	The normal animation trigger to use
Highlighted Trigger	Trigger to use when the control is highlighted
Pressed Trigger	Trigger to use when the control is pressed

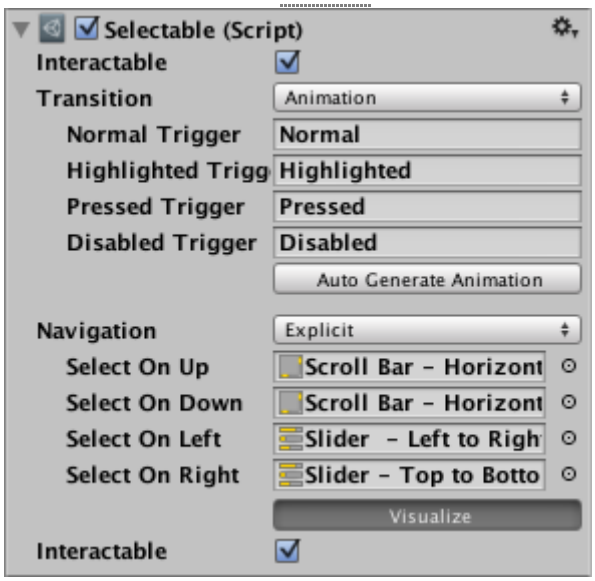
Property:	*Function:*
Disabled Trigger	Trigger to use when the control is disabled

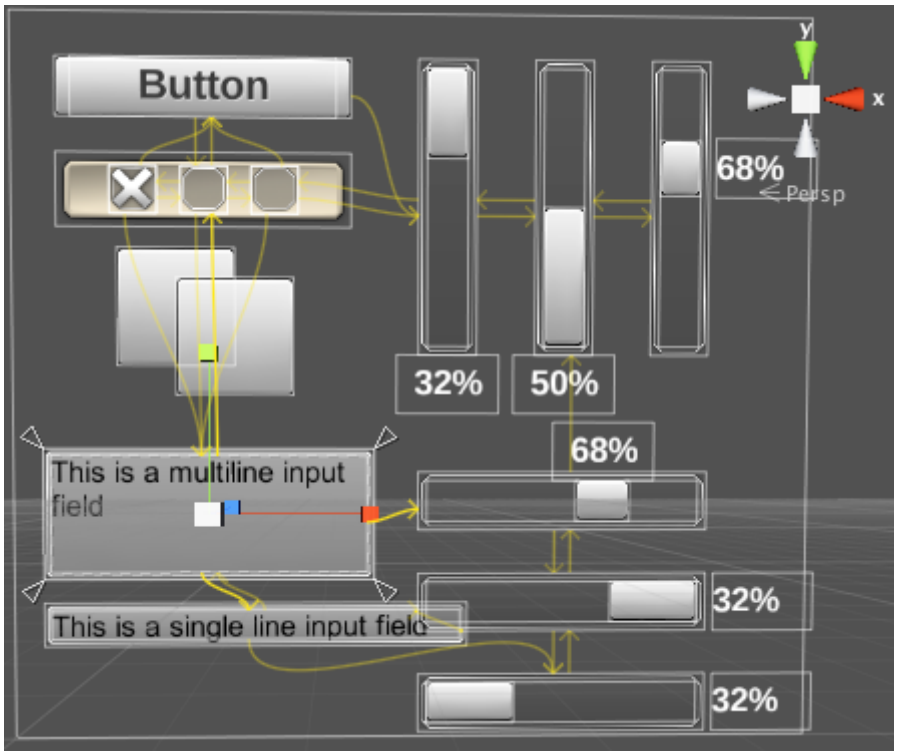
Navigation Options

Leave feedback



Property:	*Function:*
Navigation	The Navigation options refers to how the navigation of **UI** elements in play mode will be controlled.
None	No keyboard navigation. Also ensures that it does not receive focus from clicking/tapping on it.
Horizontal	Navigates Horizontally.
Vertical	Navigates Vertically.
Automatic	Automatic Navigation.
Explicit	In this mode you can explicitly specify where the control navigates to for different arrow keys.
Visualize	Selecting Visualize gives you a visual representation of the navigation you have set up in the **scene** window. See below.





Scene window showing the visualized navigation connections

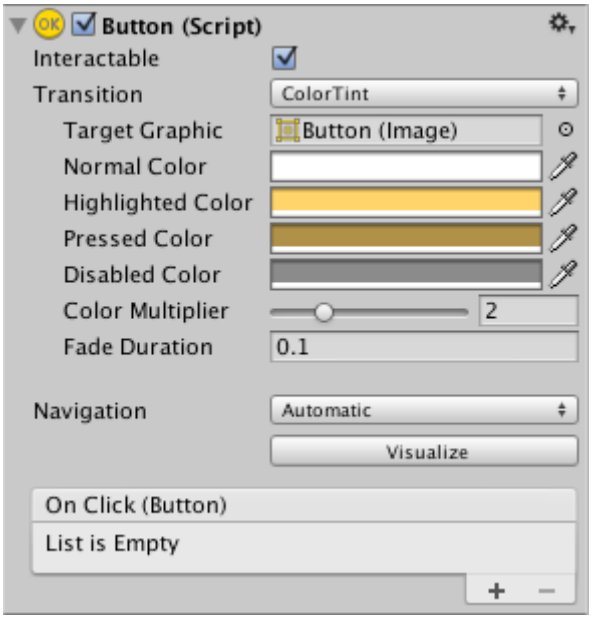
In the above visualization mode, the arrows indicate how the change of focus is set up for the collection of controls as a group. That means - for each individual UI control - you can see which UI control will get focus next, if the user presses an arrow key when the given control has focus. So in the example shown above, If the “button” has focus and the user presses the right arrow key, the first (left-hand) vertical slider will then become focused. Note that the vertical sliders can’t be focused-away-from using up or down keys, because they control the value of the slider. The same is true of the horizontal sliders and the left/right arrow keys.

Button

Leave feedback

The **Button** control responds to a click from the user and is used to initiate or confirm an action. Familiar examples include the *Submit* and *Cancel* buttons used on web forms.

A Button.



Properties

Property:	*Function:*
Interactable	Will this **component** will accept input? See Interactable.
Transition	Properties that determine the way the control responds visually to user actions. See Transition Options.
Navigation	Properties that determine the sequence of controls. See Navigation Options.

Events

Property:	*Function:*
On Click	A UnityEvent that is invoked when when a user clicks the button and releases it.

Details

The button is designed to initiate an action when the user clicks and releases it. If the mouse is moved off the button control before the click is released, the action does not take place.

The button has a single event called *On Click* that responds when the user completes a click. Typical use cases include:

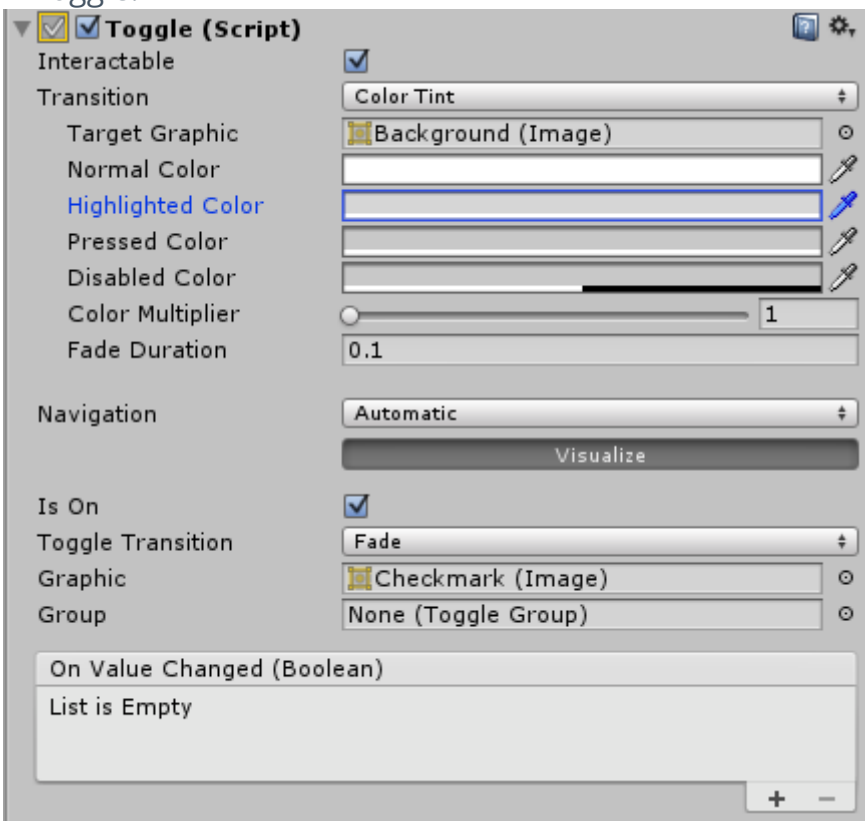
Confirming a decision (eg, starting gameplay or saving a game)
Moving to a sub-menu in a GUI
Cancelling an action in progress (eg, downloading a new scene)

Toggle

Leave feedback

The **Toggle** control is a checkbox that allows the user to switch an option on or off.

A Toggle.



Properties

Property:	*Function:*
Interactable	Will this **component** will accept input? See Interactable.
Transition	Properties that determine the way the control responds visually to user actions. See Transition Options.
Navigation	Properties that determine the sequence of controls. See Navigation Options.
Is On	Is the toggle switched on from the beginning?
Toggle Transition	The way the toggle reacts graphically when its value is changed. The options are *None* (ie, the checkmark simply appears or disappears) and *Fade* (ie, the checkmark fades in or out).
Graphic	The image used for the checkmark.
Group	The Toggle Group (if any) that this Toggle belongs to.

Events

Property:	*Function:*

Property:	*Function:*
On Value Changed	A UnityEvent that is invoked when the Toggle is clicked. The event can send the current state as a `bool` type dynamic argument.

Details

The Toggle control allows the user to switch an option on or off. You can also combine several toggles into a Toggle Group in cases where only one of a set of options should be on at once.

The Toggle has a single event called *On Value Changed* that responds when the user changes the current value. The new value is passed to the event function as a `boolean` parameter. Typical use cases for Toggles include:

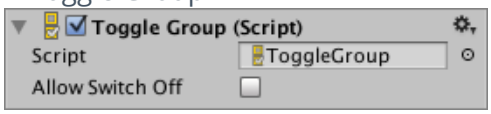
Switching an option on or off (eg, playing music during a game).
Letting the user confirm they have read a legal disclaimer.
Choosing one of a set of options (eg, a day of the week) when used in a Toggle Group.
Note that the **Toggle** is a parent that provides a clickable area to children. If the **Toggle** has no children (or they are disabled) then it is not clickable.

Toggle Group

Leave feedback

A **Toggle Group** is not a visible **UI** control but rather a way to modify the behavior of a set of Toggles. Toggles that belong to the same group are constrained so that only one of them can switched on at a time - pressing one of them to switch it on automatically switches the others off.

A Toggle Group



Properties

Property:	*Function:*
Allow Switch Off	Is it allowed that no toggle is switched on? If this setting is enabled, pressing the toggle that is currently switched on will switch it off, so that no toggle is switched on. If this setting is disabled, pressing the toggle that is currently switched on will not change its state.

Description

The Toggle Group is setup by dragging the Toggle Group object to the *Group* property of each of the Toggles in the group.

Toggle Groups are useful anywhere the user must make a choice from a mutually exclusive set of options. Common examples include selecting player character types, speed settings (slow, medium, fast, etc), preset colors and days of the week. You can have more than one Toggle Group object in the **scene** at a time, so you can create several separate groups if necessary.

Unlike other UI elements, an object with a Toggle Group **component** does not need to be a child of a Canvas object, although the Toggles themselves still do.

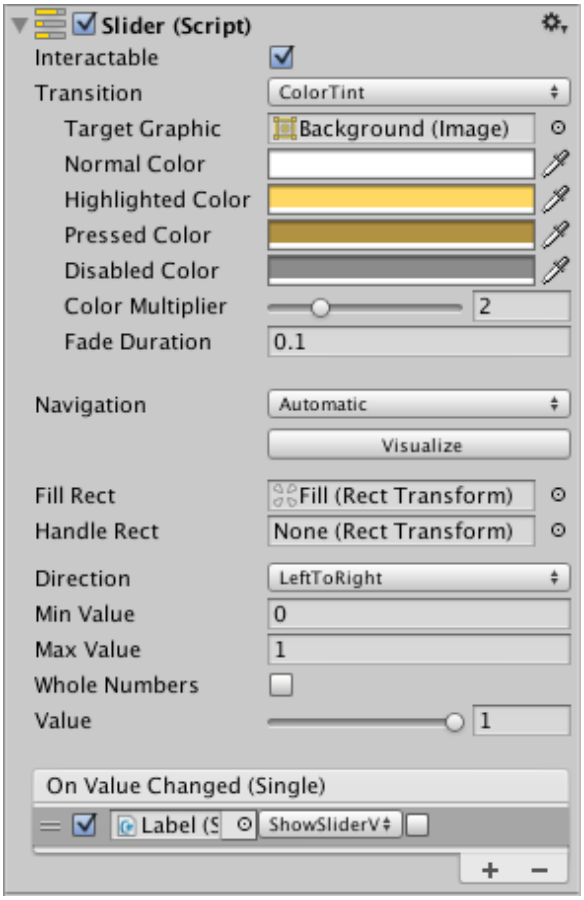
Note that the Toggle Group will not enforce its constraint right away if multiple toggles in the group are switched on when the scene is loaded or when the group is instantiated. Only when a new toggle is swicthed on are the others switched off. This means it's up to you to ensure that only one toggle is switched on from the beginning.

Slider

Leave feedback

The **Slider** control allows the user to select a numeric value from a predetermined range by dragging the mouse. Note that the similar ScrollBar control is used for scrolling rather than selecting numeric values. Familiar examples include difficulty settings in games and brightness settings in image editors.

A Slider.



Properties

Property:	*Function:*
Interactable	Will this **component** accept input? See Interactable.
Transition	Properties that determine the way the control responds visually to user actions. See Transition Options.
Navigation	Properties that determine the sequence of controls. See Navigation Options.
Fill Rect	The graphic used for the fill area of the control.
Handle Rect	The graphic used for the sliding “handle” part of the control
Direction	The direction in which the slider’s value will increase when the handle is dragged. The options are *Left To Right*, *Right To Left*, *Bottom To Top* and *Top To Bottom*.
Min Value	The value of the slider when the handle is at its extreme lower end (determined by the *Direction* property).

Property:	*Function:*
Max Value	The value of the slider when the handle is at its extreme upper end (determined by the *Direction* property).
Whole Numbers	Should the slider be constrained to integer values?
Value	Current numeric value of the slider. If the value is set in the **inspector** it will be used as the initial value, but this will change at runtime when the value changes.

Events

Property:	*Function:*
On Value Changed	A UnityEvent that is invoked when the current value of the Slider has changed. The event can send the current value as a `float` type dynamic argument. The value is passed as a float type regardless of whether the *Whole Numbers* property is enabled.

Details

The value of a Slider is determined by the position of the handle along its length. The value increases from the *Min Value* up to the *Max Value* in proportion to the distance the handle is dragged. The default behaviour is for the slider to increase from left to right but it is also possible to reverse this behavior using the *Direction* property. You can also set the slider to increase vertically by selecting *Bottom To Top* or *Top To Bottom* for the *Direction* property.

The slider has a single event called *On Value Changed* that responds as the user drags the handle. The current numeric value of the slider is passed to the function as a `float` parameter. Typical use cases include:

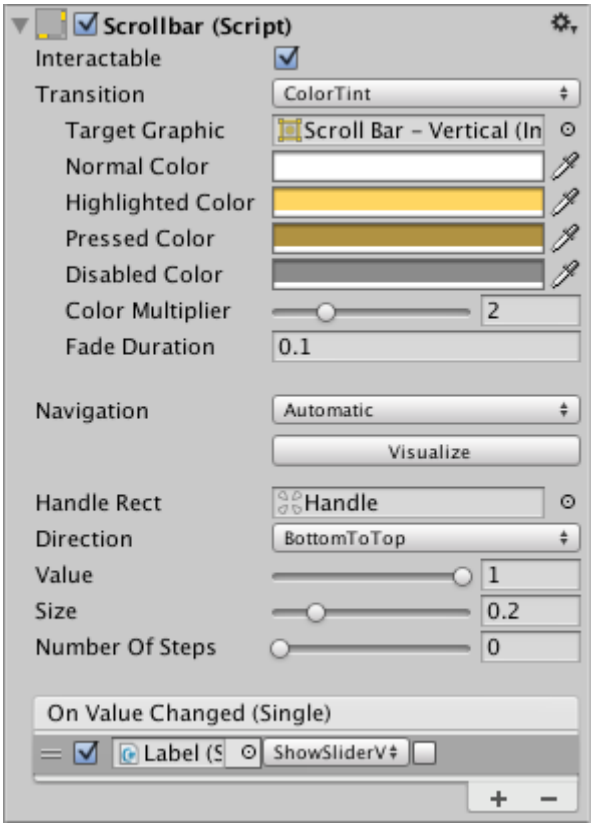
Choosing a level of difficulty in a game, brightness of a light, **etc**.
Setting a distance, size, time or angle.

Scrollbar

Leave feedback

The **Scrollbar** control allows the user to scroll an image or other view that is too large to see completely. Note that the similar Slider control is used for selecting numeric values rather than scrolling. Familiar examples include the vertical Scrollbar at the side of a text editor and the vertical and horizontal pair of bars for viewing a section of a large image or map.

A Scrollbar.



Properties

Property:	*Function:*
Interactable	Will this **component** accept input? See Interactable.
Transition	Properties that determine the way the control responds visually to user actions. See Transition Options.
Navigation	Properties that determine the sequence of controls. See Navigation Options.
Fill Rect	The graphic used for the background area of the control.
Handle Rect	The graphic used for the sliding "handle" part of the control
Direction	The direction in which the Scrollbar's value will increase when the handle is dragged. The options are *Left To Right*, *Right To Left*, *Bottom To Top* and *Top To Bottom*.
Value	Initial position value of the Scrollbar, in the range 0.0 to 1.0.
Size	Fractional size of the handle within the Scrollbar, in the range 0.0 to 1.0.

Property:	*Function:*
Number Of Steps	The number of distinct scroll positions allowed by the Scrollbar.

Events

Property:	*Function:*
On Value Changed	A UnityEvent that is invoked when the current value of the Scrollbar changes. The event can send the value as a `float` type dynamic argument.

Details

The value of a Scrollbar is determined by the position of the handle along its length with the value being reported as a fraction between the extreme ends. For example, the default left-to-right bar has a value of 0.0 at the left end, 1.0 at the right end and 0.5 indicates the halfway point. A scrollbar can be oriented vertically by choosing *Top To Bottom* or *Bottom To Top* for the *Direction* property.

A significant difference between the Scrollbar and the similar Slider control is that the Scrollbar's handle can change in size to represent the distance of scrolling available; when the view can scroll only a short way, the handle will fill up most of the bar and only allow a slight shift either direction.

The Scrollbar has a single event called *On Value Changed* that responds as the user drags the handle. The current value is passed to the even function as a `float` parameter. Typical use cases for a scrollbar include:

Scrolling a piece of text vertically.
Scrolling a **timeline** horizontally.
Used as a pair, scrolling a large image both horizontally and vertically to view a zoomed section. The size of the handle changes to indicate the degree of zooming and therefore the available distance for scrolling.

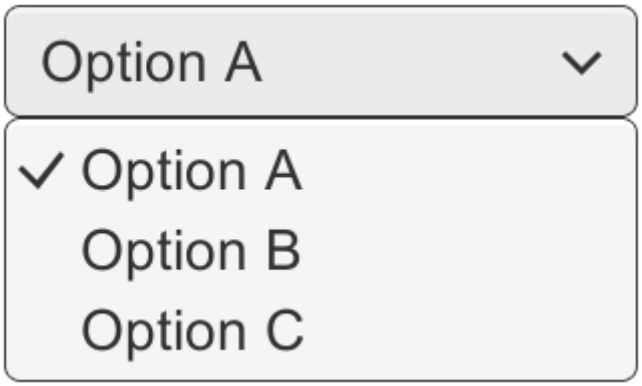
Dropdown

Leave feedback

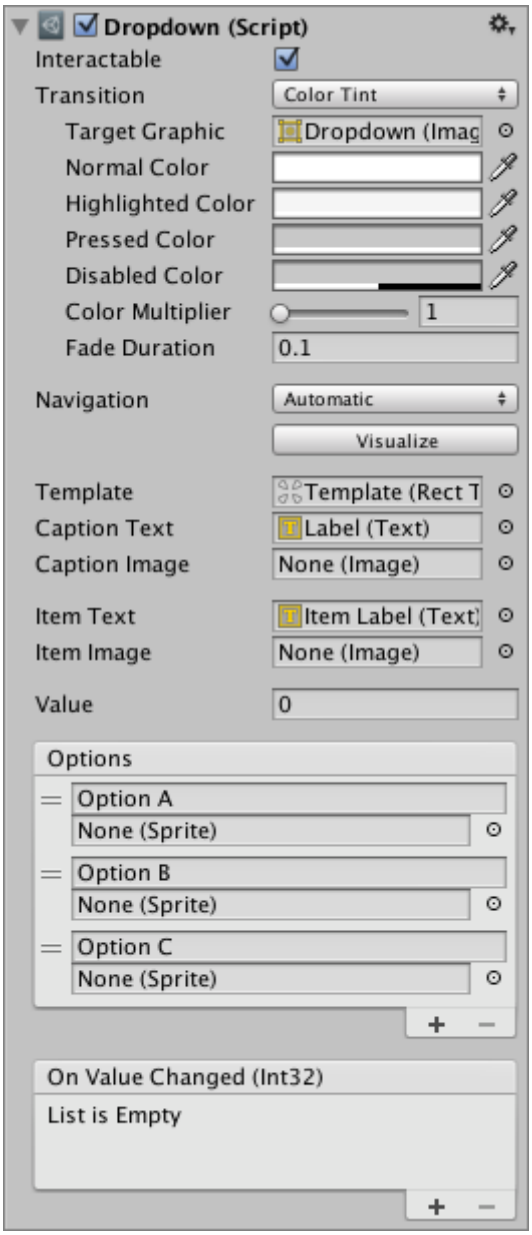
The **Dropdown** can be used to let the user choose a single option from a list of options.

The control shows the currently chosen option. Once clicked, it opens up the list of options so a new option can be chosen. Upon choosing a new option, the list of closed again, and the control shows the new selected option. The list is also closed if the user clicks on the control itself, or anywhere else inside the Canvas.

A Dropdown.



A Dropdown with its list of options open.



Properties

Property:	*Function:*
Interactable	Will this **component** will accept input? See Interactable.
Transition	Properties that determine the way the control responds visually to user actions. See Transition Options.
Navigation	Properties that determine the sequence of controls. See Navigation Options.
Template	The Rect Transform of the template for the dropdown list. See instructions below.
Caption Text	The Text component to hold the text of the currently selected option. (Optional)
Caption Image	The Image component to hold the image of the currently selected option. (Optional)
Item Text	The Text component to hold the text of the item. (Optional)
Item Image	The Image component to hold the image of the item. (Optional)
Value	The index of the currently selected option. 0 is the first option, 1 is the second, and so on.

Property:	*Function:*
Options	The list of possible options. A text string and an image can be specified for each option.

Events

Property:	*Function:*
On Value Changed	A UnityEvent that is invoked when a user has clicked one of the options in the dropdown list.

Details

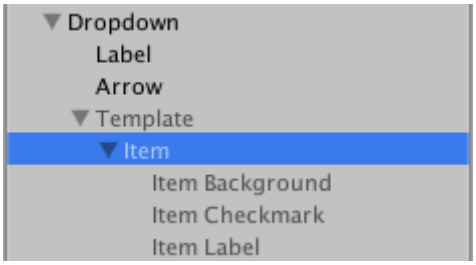
The list of options is specified in the **Inspector** or can be assigned from code. For each option a text string can be specified, and optionally an image as well, if the Dropdown is setup to support it.

The button has a single event called *On Value Changed* that responds when the user completes a click on one of the options in the list. It supports sending an integer number value that is the index of the selected option. 0 is the first option, 1 is the second, and so on.

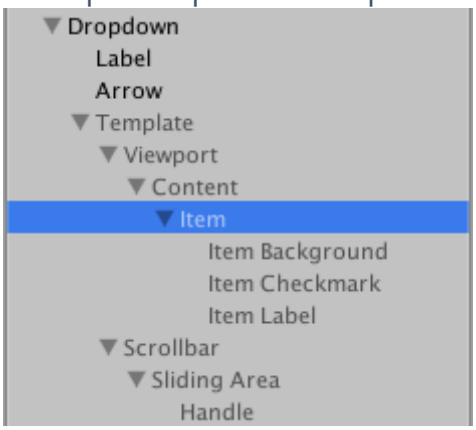
The template system

The Dropdown control is designed to have a child **GameObject** which serves as a template for the dropdown list that is shown when clicking the dropdown control. The template **GameObject** is inactive by default, but can be made active while editing the template to better see what's going on. A reference to the template object must be specified in the Template property of the Dropdown component.

The template must have a single item in it with a Toggle component on. When the actual dropdown list is created upon clicking the dropdown control, this item is duplicated multiple times, with one copy used for each option in the list. The parent of the item is automatically resized so it can fit all the items inside.



A simple dropdown setup where the item is an immediate child of the template.



A more advanced dropdown setup that includes a scrollview that enables scrolling when there are many options in the list.

The template can be setup in many different ways. The setup used by the GameObject > UI > Dropdown menu item includes a scroll view, such that if there are too many options to show at once, a scrollbar will appear and the user can scroll through the options. This is however not a mandatory part of the template setup.

(See the ScrollRect page for more information about setup of Scroll Views.)

Setup of text and image support

The dropdown supports one text content and one image content for each option. Both text and image is optional. They can only be used if the Dropdown is setup to support it.

The dropdown supports text for each option when the Caption Text and Item Text properties are both setup. These are setup by default when using the GameObject > UI > Dropdown menu item.

The Caption Text is the Text component to hold the text for the currently selected option. It is typically a child to the Dropdown GameObject.
The Item Text is the Text component to hold the text for each option. It is typically a child to the Item GameObject.
The dropdown supports an image for each option when the Caption Image and Item Image properties are both setup. These are not setup by default.

The Caption Image is the Image component to hold the image for the currently selected option. It is typically a child to the Dropdown GameObject.
The Item Image is the Image component to hold the image for each option. It is typically a child to the Item GameObject.
The actual text and images used for the dropdowns are specified in the Options property of the Dropdown component, or can be set from code.

Placement of the dropdown list

The placement of the dropdown list in relation to the dropdown control is determined by the anchoring and pivot of the Rect Transform of the Template.

By default, the list will appear below the control. This is achieved by anchoring the template to the bottom of the control. The pivot of the template also needs to be at the top, so that as the template is expanded to accommodate a variable number of option items, it only expands downwards.

The Dropdown control has simple logic to prevent that the dropdown is displayed outside the bounds of the Canvas, since this would make it impossible to select certain options. If the dropdown at its default position is not fully within the Canvas rectangle, its position in relation to the control is reversed. For example, a list that is shown below the control by default will be shown above it instead.

This logic is quite simple and has certain limitations. The dropdown template needs to be no larger than half the Canvas size minus the size of the dropdown control, otherwise there may not be room for the list at either position if the dropdown control is placed in the middle of the Canvas.

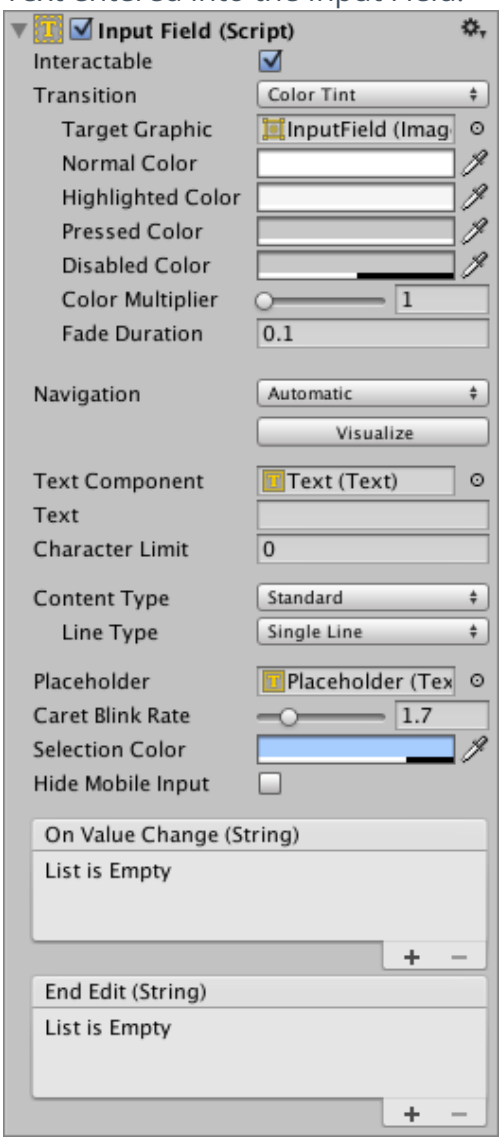
Input Field

Leave feedback

An **Input Field** is a way to make the text of a Text Control editable. Like the other interaction controls, it's not a visible **UI** element in itself and must be combined with one or more visual **UI** elements in order to be visible.

An empty Input Field.

Text entered into the Input Field.



Properties

Property:	*Function:*
Interactable	A boolean that determines if the Input Field can be interacted with or not.

Property:	*Function:*
Transition	Transitions are used to set how the input field transitions when ***Normal*** , ***Highlighted*** , ***Pressed*** or ***Disabled*** .
Navigation	Properties that determine the sequence of controls. See Navigation Options.
TextComponent	A reference to the Text element used as the contents of the *Input Field*
Text	Starting Value. The initial text placed in the field before editing begins.
Character Limit	The value of the maximum number of characters that can be entered into the input field.
Content Type	Define the type(s) of characters that your input field accepts
Standard	Any charcter can be entered.
Autocorrected	The autocorrection determines whether the input tracks unknown words and suggests a more suitable replacement candidate to the user, replacing the typed text automatically unless the user explicitly overrides the action.
Integer Number	Allow only whole numbers to be entered.
Decimal Number	Allow only numbers and a single decimal point to be entered.
Alphanumeric	Allow both letters and numbers. Symbols cannot be entered.
Name	Automatically capitalizes the first letter of each word. Note that the user can circumvent the capitalization rules using the **Delete** key.
Email Address	Allows you to enter an Alphanumeric string consisting of a maximum of one @ sign. periods/baseline dots cannot be entered next to each other.
Password*	Conceals the characters inputed with an asterisk. Allows symbols.
Pin	Conceals the characters inputed with an asterisk. Only allows only whole numbers to be entered.
Custom	Allows you to customise the Line Type, Input Type, Keyboard Type and Character Validation.
Line Type	Defines how test is formatted inside the text field.
Single Line	Only allows text to be on a single line.
Multi Line Submit	Allows text to use multiple lines. Only uses a new line when needed.
Multi Line Newline	Allows text to use multiple lines. User can use a newline by pressing the return key.
Placeholder	This is an optional 'empty' Graphic to show that the *Input Field* is empty of text. Note that this 'empty' graphic still displays even when the *Input Field* is selected (that is; when there is focus on it). eg; "Enter text...".
Caret Blink Rate	Defines the blink rate for the mark placed on the line to indicate a proposed insertion of text.
Selection Color	The background color of the selected portion of text.
Hide Mobile Input (iOS only)	Hides the native input field attached to the onscreen keyboard on mobile devices. Note that this only works on **iOS** devices.

Events

Property: Function:

Property:	*Function:*
On Value Change	A UnityEvent that is invoked when the text content of the Input Field changes. The event can send the current text content as a `string` type dynamic argument.
End Edit	A UnityEvent that is invoked when the user finishes editing the text content either by submitting or by clicking somewhere that removes the focus from the Input Field. The event can send the current text content as a `string` type dynamic argument.

Details

The Input Field script can be added to any existing Text control object from the menu (**Component > UI > Input Field**). Having done this, you should also drag the object to the Input Field's *Text* property to enable editing.

The *Text* property of the Text control itself will change as the user types and the value can be retrieved from a script after editing. Note that Rich Text is intentionally not supported for editable Text controls; the field will apply any Rich Text markup instantly when typed but the markup essentially "disappears" and there is no subsequent way to change or remove the styling.

Hints

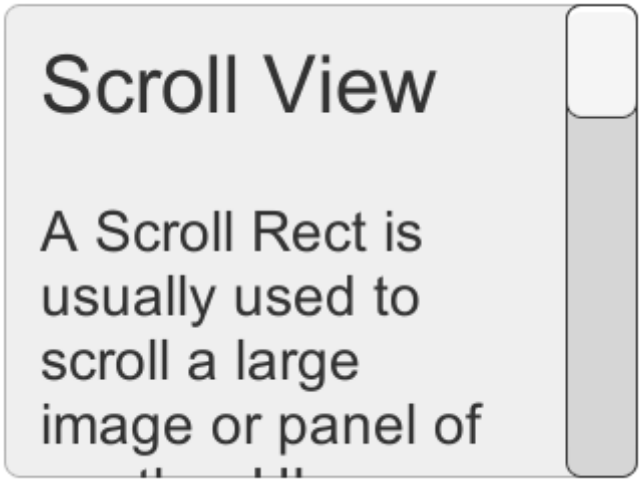
To obtains the text of the Input Field, use the text property on the InputField **component** itself, not the text property of the Text **component** that displays the text. The text property of the Text **component** may be cropped or may consist of asterisks for passwords.

Scroll Rect

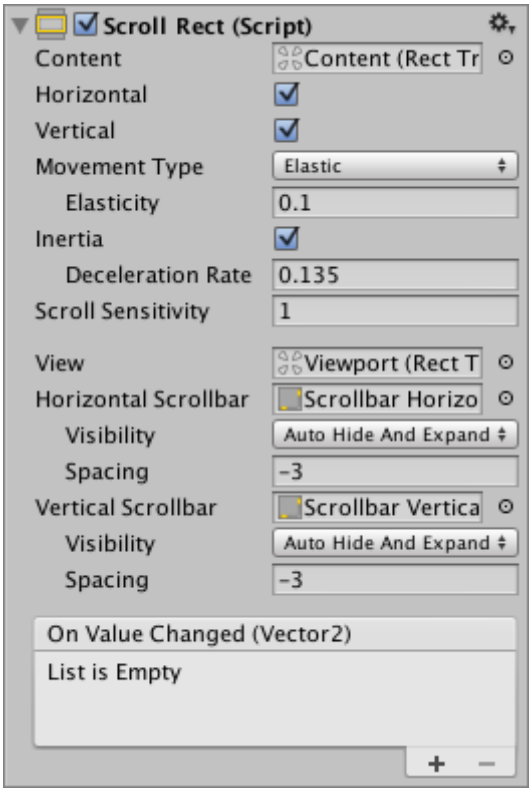
Leave feedback

A Scroll Rect can be used when content that takes up a lot of space needs to be displayed in a small area. The Scroll Rect provides functionality to scroll over this content.

Usually a Scroll Rect is combined with a Mask in order to create a scroll view, where only the scrollable content inside the Scroll Rect is visible. It can also additionally be combined with one or two Scrollbars that can be dragged to scroll horizontally or vertically.



A Scroll Rect.



Properties

Property:	*Function:*
Content	This is a reference to the Rect Transform of the **UI** element to be scrolled, for example a large image.

Property:	*Function:*
Horizontal	Enables horizontal scrolling
Vertical	Enables vertical scrolling
Movement Type	Unrestricted, Elastic or Clamped. Use Elastic or Clamped to force the content to remain within the bounds of the Scroll Rect. Elastic mode bounces the content when it reaches the edge of the Scroll Rect
Elasticity	This is the amount of bounce used in the elasticity mode.
Inertia	When Inertia is set the content will continue to move when the pointer is released after a drag. When Inertia is not set the content will only move when dragged.
Deceleration Rate	When Inertia is set the deceleration rate determines how quickly the contents stop moving. A rate of 0 will stop the movement immediately. A value of 1 means the movement will never slow down.
Scroll Sensitivity	The sensitivity to scroll wheel and track pad scroll events.
Viewport	Reference to the viewport Rect Transform that is the parent of the content Rect Transform.
Horizontal Scrollbar	Optional reference to a horizontal scrollbar element.
Visibility	Whether the scrollbar should automatically be hidden when it isn't needed, and optionally expand the viewport as well.
Spacing	The space between the scrollbar and the viewport.
Vertical Scrollbar	Optional reference to a vertical scrollbar element.
Visibility	Whether the scrollbar should automatically be hidden when it isn't needed, and optionally expand the viewport as well.
Spacing	The space between the scrollbar and the viewport.

Events

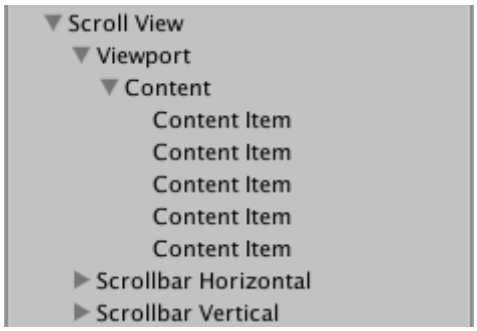
Property:	*Function:*
On Value Changed	A UnityEvent that is invoked when the scroll position of the Scroll Rect changes. The event can send the current scroll position as a `Vector2` type dynamic argument.

Details

The important elements in a scroll view are the **viewport**, the scrolling **content**, and optionally one or two **scrollbars**.

The root **GameObject** has the Scroll Rect **component**.
The viewport has a Mask component. The viewport can either be the root GameObject, or a separate GameObject that's a child to the root. If auto-hiding scrollbars are used, it must be a child. The viewport Rect Transform needs to be referenced in the **Viewport** property of the Scroll Rect.
All the scrolling content must be children of a single content GameObject that is a child to the viewport. The content Rect Transform needs to be referenced in the **Content** property of the Scroll Rect.
The scrollbars - if used - are children to the root GameObject. See the Scrollbar page for more details on the setup of a scrollbar and see the section **Scrollbar setup** below for information about

setup of scrollbars with a scroll view.
This image shows a setup where the viewport is a child to the scroll view root. This is the default used when using the GameObject > UI > Scroll View menu option.



To scroll content, the input must be received from inside the bounds of the ScrollRect, not on the content itself.

Take care when using Unrestricted scrolling movement as it is possible to lose control of the content in an irretrievable way. When using Elastic or Constrained movement it is best to position the content so that it starts within the bounds of the ScrollRect, or undesirable behaviour may occur as the RectTransform tries to bring the content back within its bounds.

Scrollbar setup

Optionally, the Scroll Rect can be linked to a horizontal and/or a vertical **Scrollbar**. These are typically placed in the hierarchy as siblings to the viewport, and when present, should be dragged into the **Horizontal Scrollbar** and **Vertical Scrollbar** properties of the Scroll Rect, respectively. Note that the **Direction** property on such a horizontal Scrollbar should be set to **Left To Right**, and on the vertical Scrollbar to **Bottom To Top**.

The scrollbars can optionally have auto-hiding behaviour that hides the scrollbars if the content doesn't need to scroll because it isn't larger than the viewport. Note that the auto-hiding only ever happens in Play Mode. In Edit Mode the scrollbars are always shown. This prevents marking the **scene** as dirty when it shouldn't be, and also help authoring content with proportions that there's room for even when the scrollbars are shown.

If one or both scrollbars have their visibility behaviour set to **Auto Hide And Expand View**, the viewport is automatically expanded when the scrollbars are hidden in order to take up the extra room where the scrollbars would otherwise have been. With this setup, the position and size of the **view** is driven by the Scroll Rect, and the width of the **horizontal scrollbar** as well as the height of the **vertical scrollbar** is driven as well. With this setup the viewport as well as the scrollbars must be children to the Scroll Rect root GameObject.

Hints

The pivot and anchors of the content RectTransform can be used to determine how the content is aligned inside the scroll view if the content grows or shrinks. If the content should stay aligned with the top, set the anchors to the top of the parent, and set the pivot to the top position.
See the page Making UI elements fit the size of their content for information about how to make the content Rect Transform automatically resize to fit the content.

Auto Layout

Leave feedback

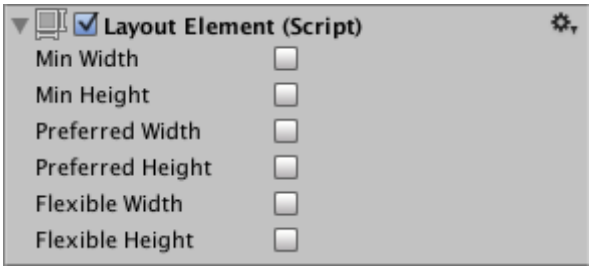
The auto layout system provides ways to place elements in nested layout groups such as horizontal groups, vertical groups, or grids. It also allows elements to automatically be sized according to the contained content.

Content Size Fitter
Layout Element
Horizontal Layout Group
Vertical Layout Group
Grid Layout Group

Layout Element

Leave feedback

Properties



Property:	*Function:*
Min Width	The minimum width this layout element should have.
Min Height	The minimum height this layout element should have.
Preferred Width	The preferred width this layout element should have before additional available width is allocated.
Preferred Height	The preferred height this layout element should have before additional available height is allocated.
Flexible Width	The relative amount of additional available width this layout element should fill out relative to its siblings.
Flexible Height	The relative amount of additional available height this layout element should fill out relative to its siblings.

Description

If you want to override the minimum, preferred, or flexible size of a layout element, you can do that by adding a Layout Element **component** to the Game Object.

The properties are used in the following manner when a layout controller allocates width or height to a layout element:

First minimum sizes are allocated.
If there is sufficient available space, preferred sizes are allocated.
If there is additional available space, flexible size is allocated.

The Layout Element component lets you override the values for one or more of the layout properties. Enable the checkbox for a property you want to override and then specify the value you want to override with.

Minimum and preferred sizes are defined in regular units, while the flexible sizes are defined in relative units. If any layout element has flexible size greater than zero, it means that all the available space will be filled out. The relative flexible size values of the siblings determines how big a proportion of the available space each sibling fills out. Most commonly, flexible width and height is set to just 0 or 1.

Specifying both a preferred size and a flexible size can make sense in certain cases. Flexible sizes are only allocated after all preferred sizes have been fully allocated. Thus, a layout element which has a flexible size specified but no preferred size will keep its minimum size until other layout elements have grown to their full preferred size, and only then begin to grow based on additional available space. By also specifying a flexible size, this can be avoided and the element can grow to its preferred size in tandem with the other layout elements that have preferred sizes, and then grow further once all flexible sizes have been allocated.

Content Size Fitter

Leave feedback

Properties

Property:	*Function:*
Horizontal Fit	How the width is controlled.
Unconstrained	Do not drive the width based on the layout element.
Min Size	Drive the width based on the minimum width of the layout element.
Preferred Size	Drive the width based on the preferred width of the layout element.
Vertical Fit	How the height is controlled.
Unconstrained	Do not drive the height based on the layout element.
Min Size	Drive the height based on the minimum height of the layout element.
Preferred Size	Drive the height based on the preferred height of the layout element.

Description

The Content Size Fitter functions as a layout controller that controls the size of its own layout element. The size is determined by the minimum or preferred sizes provided by layout element components on the Game Object. Such layout elements can be Image or Text components, layout groups, or a Layout Element **component**.

It's worth keeping in mind that when a Rect Transform is resized - whether by a Content Size Fitter or something else - the resizing is around the pivot. This means that the direction of the resizing can be controlled using the pivot.

For example, when the pivot is in the center, the Content Size Fitter will expand the Rect Transform out equally in all directions. And when the pivot is in the upper left corner, the Content Size Fitter will expand the Rect Transform down and to the right.

Aspect Ratio Fitter

Leave feedback

Properties

Property:	*Function:*
Aspect Mode	How the rectangle is resized to enforce the aspect ratio.
None	Do not make the rect fit the aspect ratio.
Width Controls Height	The height is automatically adjusted based on the width.
Height Controls Width	The width is automatically adjusted based on the height.
Fit In Parent	The width, height, position, and anchors are automatically adjusted to make the rect fit inside the rect of the parent while keeping the aspect ratio. The may be some space inside the parent rect which is not covered by this rect.
Envelope Parent	The width, height, position, and anchors are automatically adjusted to make the rect cover the entire area of the parent while keeping the aspect ratio. This rect may extend further out than the parent rect.
Aspect Ratio	The aspect ratio to enforce. This is the width divided by the height.

Description

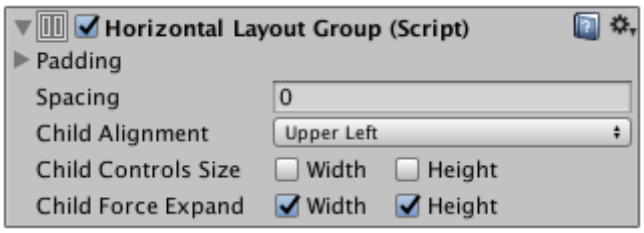
The Aspect Ratio Fitter functions as a layout controller that controls the size of its own layout element. It can adjust the height to fit the width or vice versa, or it can make the element fit inside its parent or envelope its parent. The Aspect Ratio Fitter does not take layout information into account such as minimum size and preferred size.

It's worth keeping in mind that when a Rect Transform is resized - whether by an Aspect Ratio Fitter or something else - the resizing is around the pivot. This means that the pivot can be used to control the alignment of the rectangle. For example, a pivot placed at the top center will make the rectangle grow evenly to both sides, and only grow downwards while the top edge remain at its position.

Horizontal Layout Group

Leave feedback

Properties



Property:	*Function:*
Padding	The padding inside the edges of the layout group.
Spacing	The spacing between the layout elements.
Child Alignment	The alignment to use for the child layout elements if they don't fill out all the available space.
Child Controls Size	Whether the Layout Group controls the width and height of its children.
Child Force Expand	Whether to force the children to expand to fill additional available space.

Description

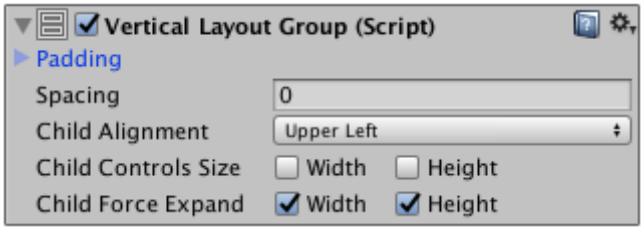
The Horizontal Layout Group **component** places its child layout elements next to each other, side by side. Their widths are determined by their respective minimum, preferred, and flexible widths according to the following model:

The minimum widths of all the child layout elements are added together and the spacing between them is added as well. The result is the mimimum width of the Horizontal Layout Group.
The preferred widths of all the child layout elements are added together and the spacing between them is added as well. The result is the preferred width of the Horizontal Layout Group.
If the Horizontal Layout Group is at its minimum width or smaller, all the child layout elements will also have their minimum width.
The closer the Horizontal Layout group is to its preferred width, the closer each child layout element will also get to their preferred width.
If the Horizontal Layout Group is wider than its preferred width, it will distribute the extra available space proportionally to the child layout elements according to their respective flexible widths.

Vertical Layout Group

Leave feedback

Properties



Property:	*Function:*
Padding	The padding inside the edges of the layout group.
Spacing	The spacing between the layout elements.
Child Alignment	The alignment to use for the child layout elements if they don't fill out all the available space.
Child Controls Size	Whether the Layout Group controls the width and height of its children.
Child Force Expand	Whether to force the children to expand to fill additional available space.

Description

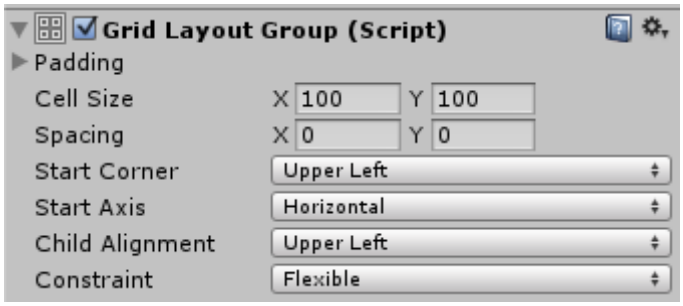
The Vertical Layout Group **component** places its child layout elements on top of each other. Their heights are determined by their respective minimum, preferred, and flexible heights according to the following model:

The minimum heights of all the child layout elements are added together and the spacing between them is added as well. The result is the mimimum height of the Vertical Layout Group.
The preferred heights of all the child layout elements are added together and the spacing between them is added as well. The result is the preferred height of the Vertical Layout Group.
If the Vertical Layout Group is at its minimum height or smaller, all the child layout elements will also have their minimum height.
The closer the Vertical Layout group is to its preferred height, the closer each child layout element will also get to their preferred height.
If the Vertical Layout Group is taller than its preferred height, it will distribute the extra available space proportionally to the child layout elements according to their respective flexible heights.

Grid Layout Group

Leave feedback

The Grid Layout Group **component** places its child layout elements in a grid.



Properties

Property:	*Function:*
Padding	The padding inside the edges of the layout group.
Cell Size	The size to use for each layout element in the group.
Spacing	The spacing between the layout elements.
Start Corner	The corner where the first element is located.
Start Axis	Which primary axis to place elements along. Horizontal will fill an entire row before a new row is started. Vertical will fill an entire column before a new column is started.
Child Alignment	The alignment to use for the layout elements if they don't fill out all the available space.
Constraint	Constraint the grid to a fixed number of rows or columns to aid the auto layout system.

Description

Unlike other layout groups, the Grid Layout Group ignores the minimum, preferred, and flexible size properties of its contained layout elements and instead assigns a fixed size to all of them which is defined with the Cell Size property of the Grid Layout Group itself.

Grid Layout Group and auto layout

There are special considerations to be aware of when using the Grid Layout Group as part of an auto layout setup, such as using it with a Content Size Fitter.

The auto layout system calculates the horizontal and vertical sizes independently. This can be at odds with the Grid Layout Group, where the number of rows depends on the number of columns and vice versa.

For any given number of cells, there are different combinations of row count and column count that can make the grid fit its content. In order to aid the layout system, you can specify that you intent the table to have a fixed number of columns or rows by using the Constraint property.

Here are suggested ways of using the Layout System with a Content Size Fitter:

Flexible width and fixed height

To setup a grid with a flexible width and fixed height, where the grid expands horizontally as more elements are added, you can set these properties as follows:

Grid Layout Group Constraint: Fixed Row Count
Content Size Fitter Horizontal Fit: Preferred Size
Content Size Fitter Vertical Fit: Preferred Size or Unconstrained

If unconstrained Vertical Fit is used, it's up to you to give the grid a height that is big enough to fit the specified row count of cells.

Fixed width and flexible height

To setup a grid with a fixed width and flexible height, where the grid expands vertically as more elements are added, you can set these properties as follows:

Grid Layout Group Constraint: Fixed Column Count
Content Size Fitter Horizontal Fit: Preferred Size or Unconstrained
Content Size Fitter Vertical Fit: Preferred Size

If unconstrained Horizontal Fit is used, it's up to you to give the grid a width that is big enough to fit the specified column count of cells.

Both flexible width and height

If you want a grid with both a flexible width and height you can do that, but you will have no control over the specific number of rows and columns. The grid will attempt to make the row and column count approximately the same. You can set these properties as follows:

Grid Layout Group Constraint: Flexible
Content Size Fitter Horizontal Fit: Preferred Size
Content Size Fitter Vertical Fit: Preferred Size

UI How Tos

Leave feedback

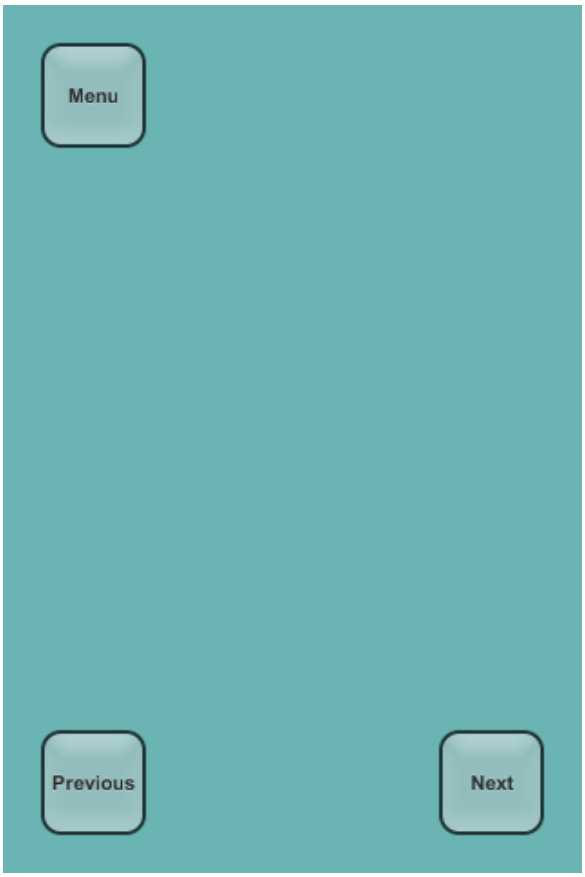
In this section you can learn about solutions to common **UI** tasks.

Designing UI for Multiple Resolutions

Leave feedback

Modern games and applications often need to support a wide variety of different screen resolutions and particularly **UI** layouts need to be able to adapt to that. The **UI** System in Unity includes a variety of tools for this purpose that can be combined in various ways.

In this how-to we're going to use a simple case study and look at and compare the different tools in the context of that. In our case study we have three buttons in the corners of the screen as shown below, and the goal is to adapt this layout to various resolutions.

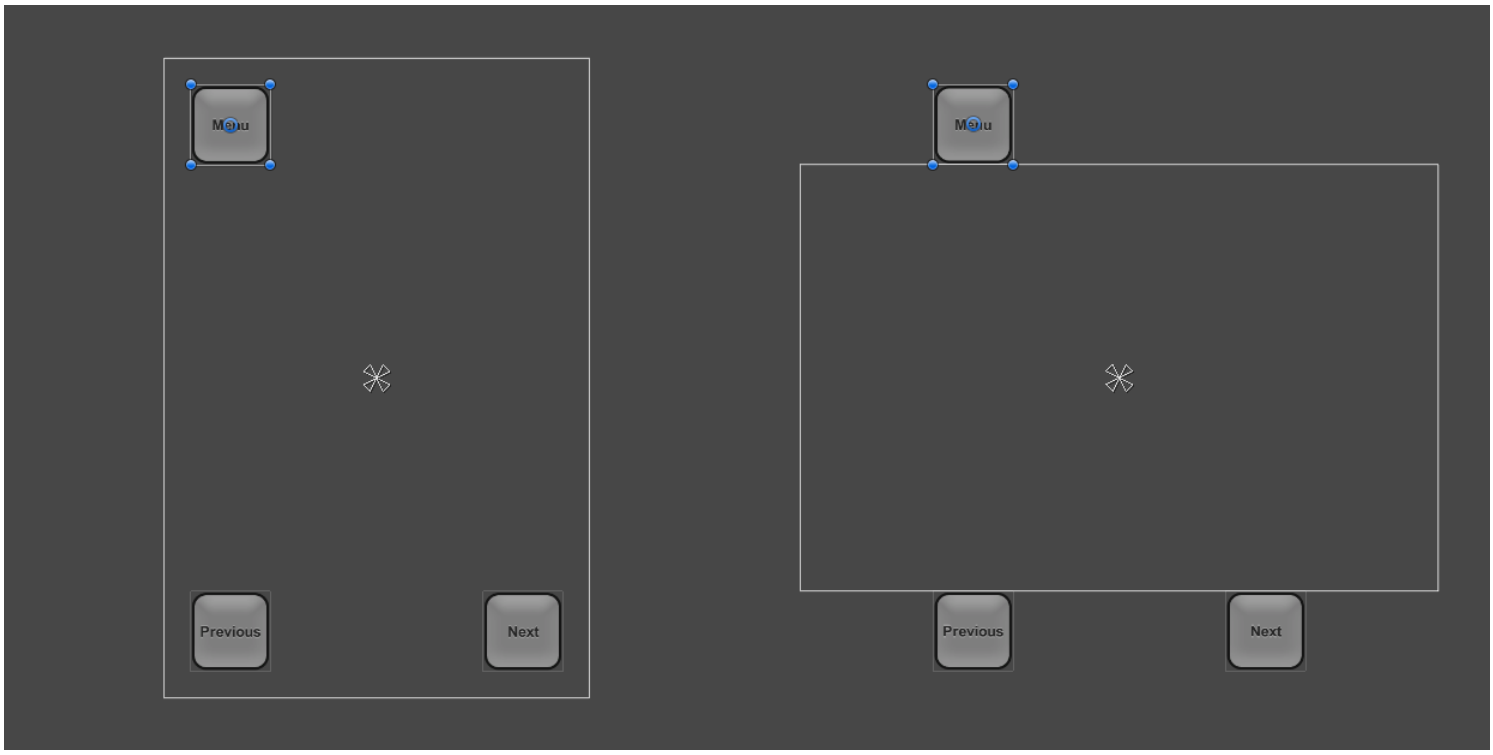


For this how-to we're going to consider four screen resolutions: Phone HD in portrait (640 x 960) and landscape (960 x 640) and Phone SD in portrait (320 x 480) and landscape (480 x 320). The layout is initially setup in the Phone HD Portrait resolution.

Using anchors to adapt to different aspect ratios

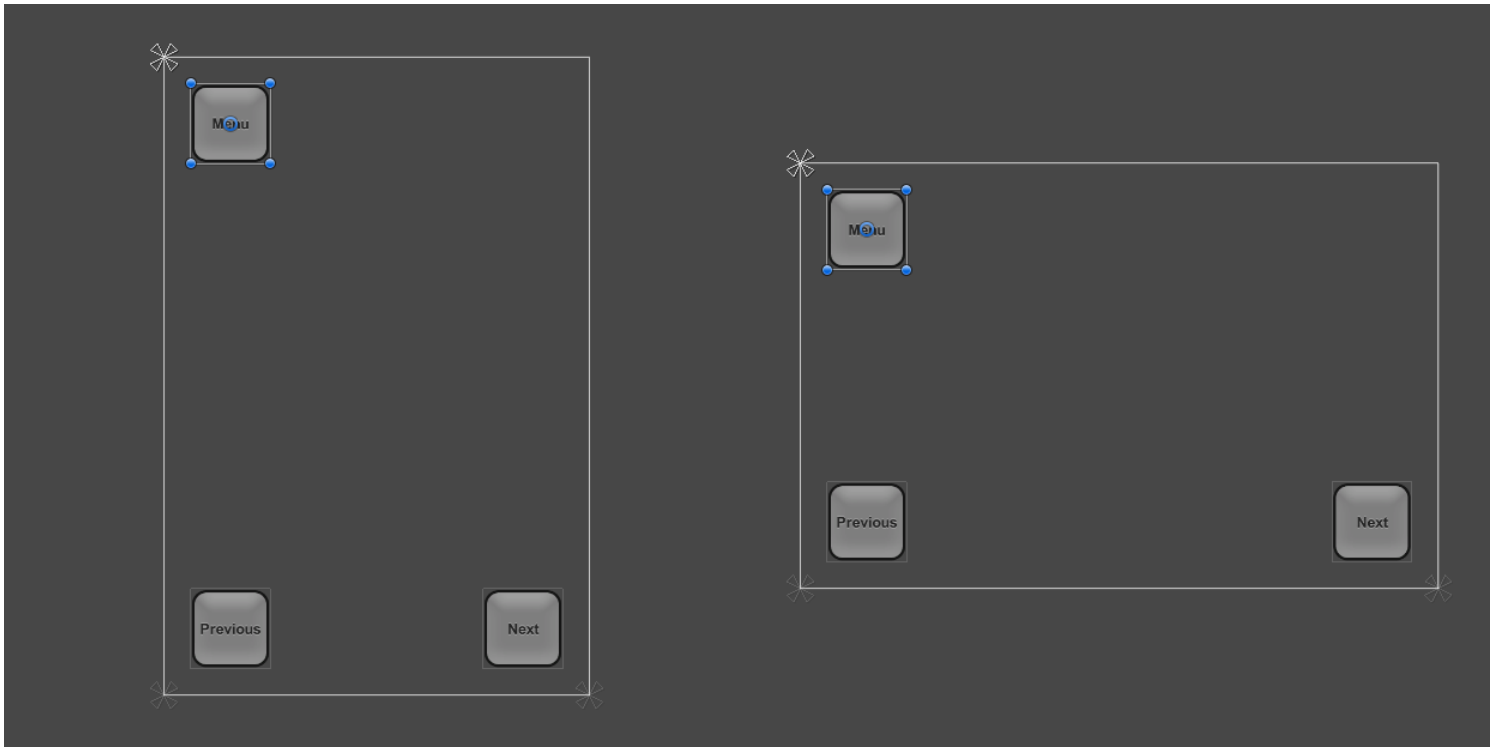
UI elements are by default anchored to the center of the parent rectangle. This means that they keep a constant offset from the center.

If the resolution is changed to a landscape **aspect ratio** with this setup, the buttons may not even be inside the rectangle of the screen anymore.



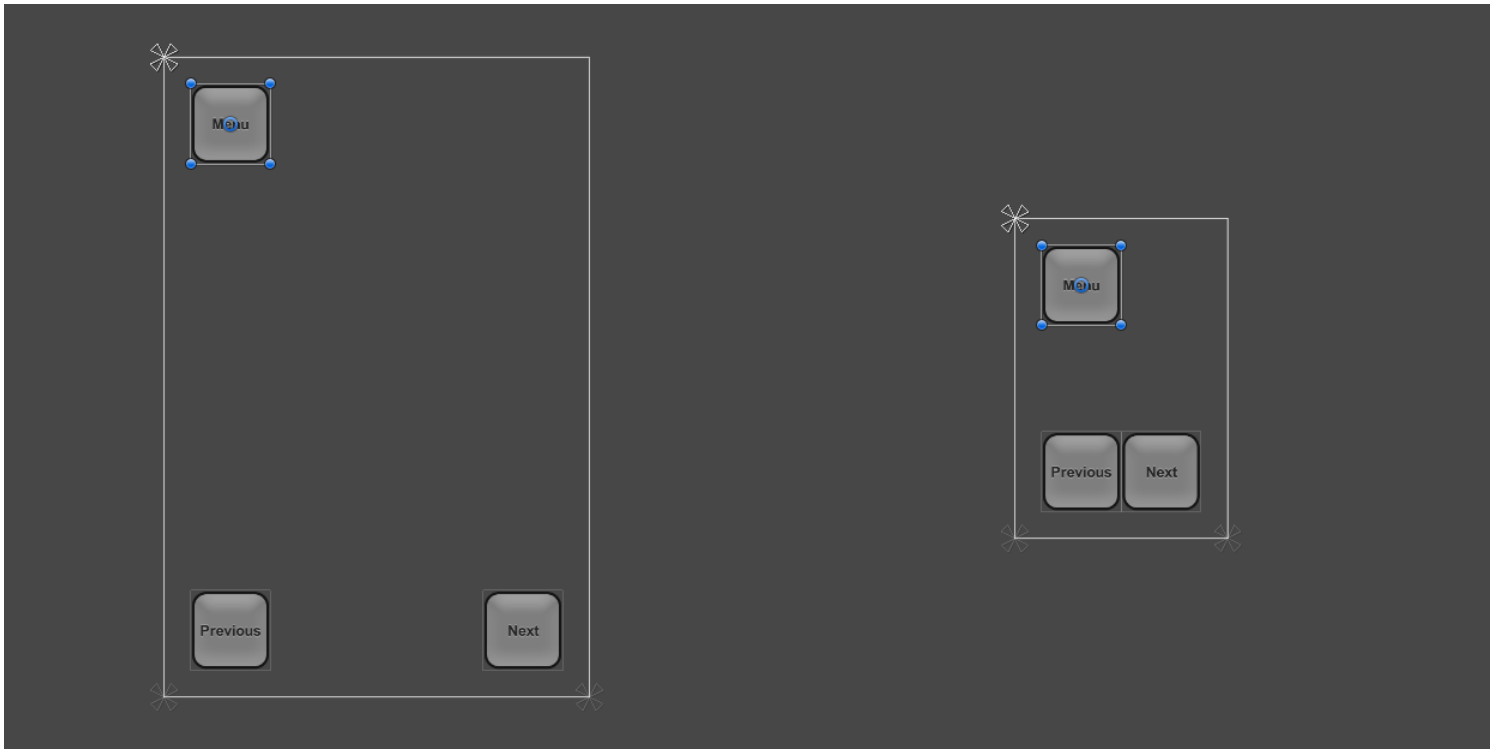
One way to keep the buttons inside the screen is to change the layout such that the locations of the buttons are tied to their respective corners of the screen. The anchors of the top left button can be set to the upper left corner using the Anchors Preset drop down in the **Inspector**, or by dragging the triangular anchor handles in the **Scene View**. It's best to do this while the current screen resolution set in the Game View is the one the layout is initially designed for, where the button placement looks correct. (See the UI Basic Layout page for more information on anchors.) Similarly, the anchors for the lower left and lower right buttons can be set to the lower left corner and lower right corner, respectively.

Once the buttons have been anchored to their respective corners, they stick to them when changing the resolution to a different aspect ratio.



When the screen size is changed to a larger or smaller resolution, the buttons will also remain anchored to their respective corners. However, since they keep their original size as specified in **pixels**, they may take up a larger or smaller proportion of

the screen. This may or may not be desirable, depending on how you would like your layout to behave on screens of different resolutions.



In this how-to, we know that the smaller resolutions of Phone SD Portrait and Landscape don't correspond to screens that are physically smaller, but rather just screens with a lower pixel density. On these lower-density screens the buttons shouldn't appear larger than on the high-density screens - they should instead appear with the same size.

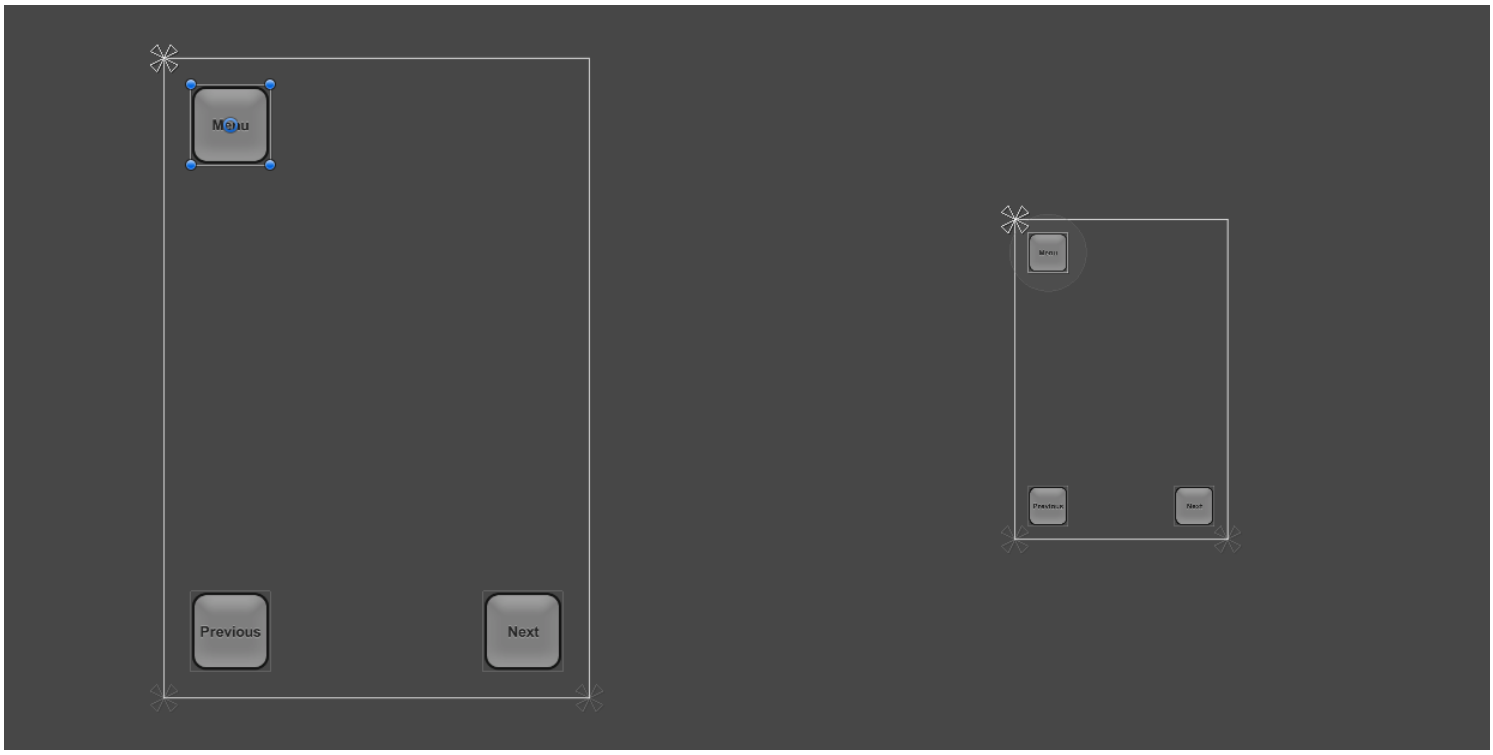
This means that the buttons should become smaller by the same percentage as the screen is smaller. In other words, the scale of the buttons should follow the screen size. This is where the **Canvas Scaler component** can help.

Scaling with Screen Size

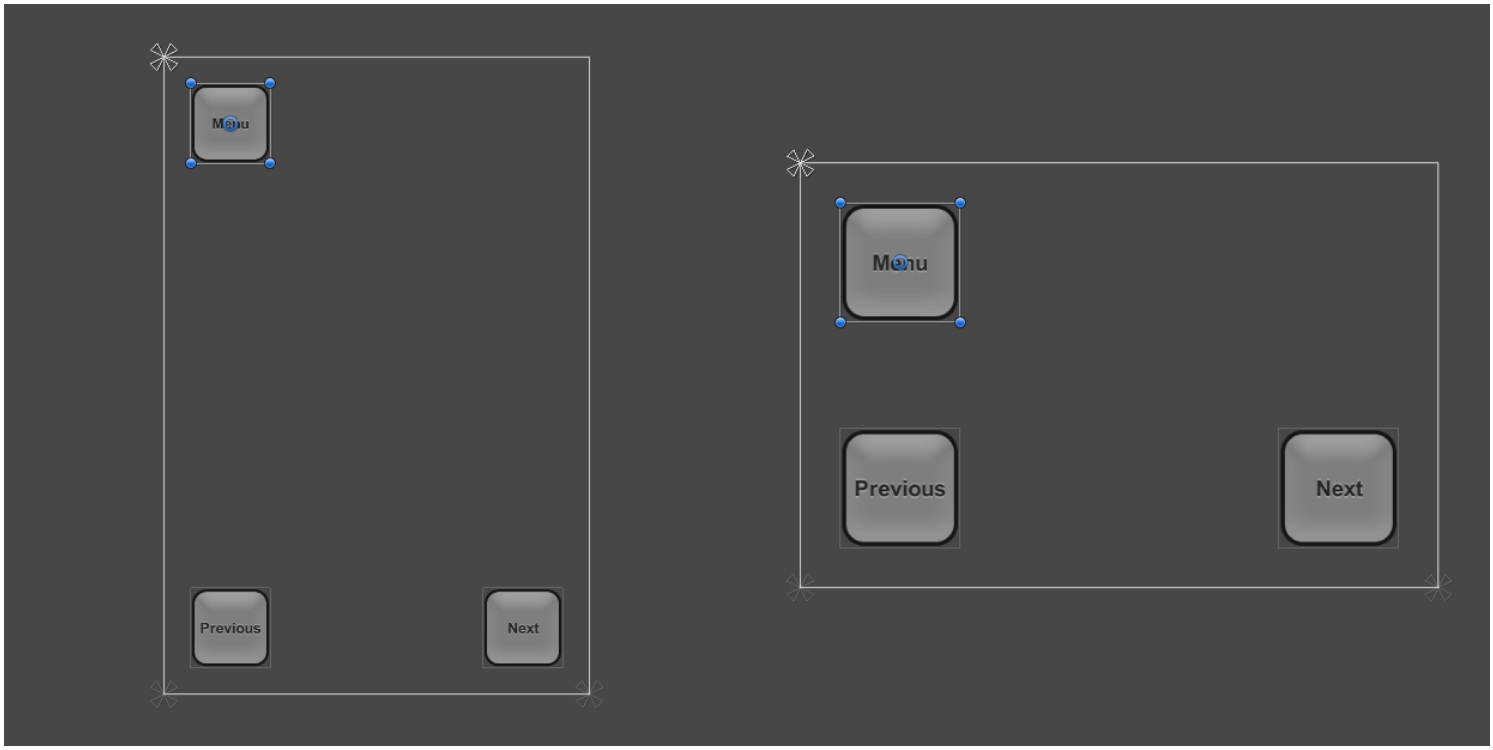
The **Canvas Scaler** component can be added to a root **Canvas** - a Game Object with a Canvas component on it, which all the UI elements are children of. It is also added by default when creating a new Canvas through the **GameObject** menu.

In the Canvas Scaler component, you can set its **UI Scale Mode** to **Scale With Screen Size**. With this scale mode you can specify a resolution to use as reference. If the current screen resolution is smaller or larger than this reference resolution, the scale factor of the Canvas is set accordingly, so all the UI elements are scaled up or down together with the screen resolution.

In our case, we set the **Canvas Scaler** to be the Phone HD portrait resolution of 640 x 960. Now, when setting the screen resolution to the Phone SD portrait resolution of 320 x 480, the entire layout is scaled down so it appears proportionally the same as in full resolution. Everything is scaled down: The button sizes, their distances to the edges of the screen, the button graphics, and the text elements. This means that the layout will appear the same in the Phone SD portrait resolution as in Phone HD portrait; only with a lower pixel density.



One thing to be aware of: After adding a Canvas Scaler component, it's important to also check how the layout looks at other aspect ratios. By setting the resolution back to Phone HD landscape, we can see that the buttons now appear bigger than they should (and used to).

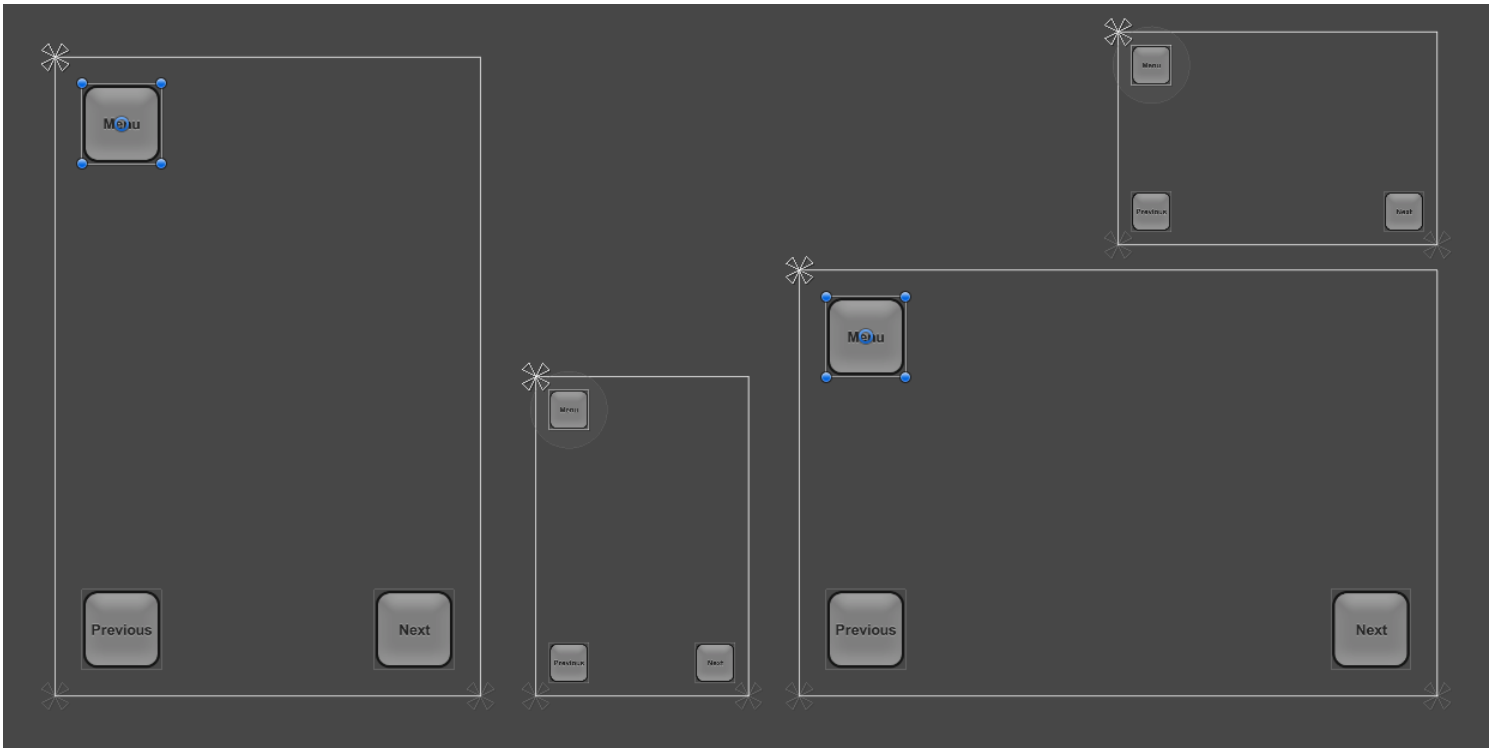


The reason for the larger buttons in landscape aspect ratio comes down to how the Canvas Scaler setting works. By default it compares the width or the current resolution with the width of the Canvas Scaler and the result is used as the scale factor to scale everything with. Since the current landscape resolution of 960 x 640 has a 1.5 times larger width than the portrait Canvas Scaler of 640 x 960, the layout is scaled up by 1.5.

The component has a property called **Match** which can be 0 (Width), 1 (Height) or a value in between. By default it's set to 0, which compares the current screen width with the Canvas Scaler width as described.

If the **Match** property is set to 0.5 instead, it will compare both the current width to the reference width and the current height to the reference height, and choose a scale factor that's in between the two. Since in this case the landscape resolution is 1.5 times wider but also 1.5 times shorter, those two factor even out and produce a final scale factor of 1, which means the buttons keep their original size.

At this point the layout supports all the four screen resolutions using a combination of appropriate anchoring and the Canvas Scaler component on the Canvas.



See the Canvas Scaler reference page for more information on different ways to scale UI elements in relation to different screen sizes.

Making UI elements fit the size of their content

Leave feedback

Normally when positioning a **UI** element with its Rect Transform, its position and size is specified manually (optionally including behavior to stretch with the parent Rect Transform).

However, sometimes you may want the rectangle to be automatically sized to fit the content of the UI element. This can be done by adding a **component** called Content Size Fitter.

Fit to size of Text

In order to make a Rect Transform with a Text component on it fit the text content, add a Content Size Fitter component to the same Game Object which has the Text component. Then set both the Horizontal Fit and Vertical Fit dropdowns to the Preferred setting.

How does it work?

What happens here is that the Text component functions as a Layout Element that can provide information about how big its minimum and preferred size is. In a manual layout this information is not used. A Content Size Fitter is a type of Layout Controller, which listens to layout information provided by Layout Elements and control the size of the Rect Transform according to this.

Remember the pivot

When UI elements are automatically resized to fit their content, you should pay extra attention to the pivot of the Rect Transform. The pivot will stay in place when the element is resized, so by setting the pivot position you can control in which direction the element will expand or shrink. For example, if the pivot is in the center, then the element will expand equally in all directions, and if the pivot is in the upper left corner, then the element will expand to the right and down.

Fit to size of UI element with child Text

If you have a UI element, such as a Button, that has a background image and a child Game Object with a Text component on it, you probably want the whole UI element to fit the size of the text - maybe with some padding.

In order to do this, first add a Horizontal Layout Group to the UI element, then add a Content Size Fitter too. Set the Horizontal Fit, the Vertical Fit, or both to the Preferred setting. You can add and tweak padding using the padding property in the Horizontal Layout Group.

Why use a Horizontal Layout Group? Well, it could have been a Vertical Layout Group as well - as long as there is only a single child, they produce the same result.

How does it work?

The Horizontal (or Vertical) Layout Group functions both as a Layout Controller and as a Layout Element. First it listens to the layout information provided by the children in the group - in this case the child Text. Then it determines how large the group must be (at minimum, and preferably) in order to be able to contain all the children, and it functions as a Layout Element that provides this information about its minimum and preferred size.

The Content Size Fitter listens to layout information provided by any Layout Element on the same Game Object - in this case provided by the Horizontal (or Vertical) Layout Group. Depending on its settings, it then controls the size of the Rect Transform based on this information.

Once the size of the Rect Transform has been set, the Horizontal (or Vertical) Layout Group makes sure to position and size its children according to the available space. See the page about the Horizontal Layout Group for more information about how it controls the positions and sizes of its children.

Make children of a Layout Group fit their respective sizes

If you have a Layout Group (horizontal or vertical) and want each of the UI elements in the group to fit their respective content, what do you do?

You can't put a Content Size Fitter on each child. The reason is that the Content Size Fitter wants control over its own Rect Transform, but the parent Layout Group also wants control over the child Rect Transform. This creates a conflict and the result is undefined behavior.

However, it isn't necessary either. The parent Layout Group can already make each child fit the size of the content. What you need to do is to disable the Child Force Expand toggles on the Layout Group. If the children are themselves Layout Groups too, you may need to disable the Child Force Expand toggles on those too.

Once the children no longer expand with flexible width, their alignment can be specified in the Layout Group using the Child Alignment setting.

What if you want some of the children to expand to fill additional available space, but not the other children? You can easily control this by adding a Layout Element component to the children you want to expand and enabling the Flexible Width or Flexible Height properties on those Layout Elements. The parent Layout Group should still have the Child Force Expand toggles disabled, otherwise all the children will expand flexibly.

How does it work?

A Game Object can have multiple components that each provide layout information about minimum, preferred and flexible sizes. A priority system determines which values take effect over others. The Layout Element component has a higher priority than the Text, Image, and Layout Group components, so it can be used to override any layout information values they provide.

When the Layout Group listens to the layout information provided by the children, it will take the overridden flexible sizes into account. Then, when controlling the sizes of the children, it will not make them any bigger than their preferred sizes. However, if the Layout Group has the Child Force Expand option enabled, it will always make the flexible sizes of all the children be at least 1.

More information

This page has explained solutions to a few common use cases. For a more in depth explanation of the auto layout system, see the UI Auto Layout page.

Creating a World Space UI

Leave feedback

The **UI** system makes it easy to create **UI** that is positioned in the world among other 2D or **3D objects** in the **Scene**.

Start by creating a UI element (such as an Image) if you don't already have one in your scene by using GameObject > UI > Image. This will also create a Canvas for you.

Set the Canvas to World Space

Select your Canvas and change the Render Mode to World Space.

Now your Canvas is already positioned in the World and can be seen by all **cameras** if they are pointed at it, but it is probably huge compared to other objects in your Scene. We'll get back to that.

Decide on a resolution

First you need to decide what the resolution of the Canvas should be. If it was an image, what should the **pixel** resolution of the image be? Something like 800x600 might be a good starting point. You enter the resolution in the Width and Height values of the Rect Transform of the Canvas. It's probably a good idea to set the position to 0,0 at the same time.

Specify the size of the Canvas in the world

Now you should consider how big the Canvas should be in the world. You can use the Scale tool to simply scale it down until it has a size that looks good, or you can decide how big it should be in meters.

If you want it to have a specific width in meters, you can can calculate the needed scale by using meter_size / canvas_width. For example, if you want it to be 2 meters wide and the Canvas width is 800, you would have 2 / 800 = 0.0025. You then set the Scale property of the Rect Transform on the Canvas to 0.0025 for both X, Y, and Z in order to ensure that it's uniformly scaled.

Another way to think of it is that you are controlling the size of one pixel in the Canvas. If the Canvas is scaled by 0.0025, then that is also the size in the world of each pixel in the Canvas.

Position the Canvas

Unlike a Canvas set to Screen Space, a World Space Canvas can be freely positioned and rotated in the Scene. You can put a Canvas on any wall, floor, ceiling, or slanted surface (or hanging freely in the air of course). Just use the normal Translate and Rotate tools in the **toolbar**.

Create the UI

Now you can begin setting up your UI elements and layouts the same way you would with a Screen Space Canvas.

Creating UI elements from scripting

Leave feedback

If you are creating a dynamic **UI** where **UI** elements appear, disappear, or change based on user actions or other actions in the game, you may need to make a script that instantiates new **UI** elements based on custom logic.

Creating a prefab of the UI element

In order to be able to easily instantiate UI elements dynamically, the first step is to create a **prefab** for the type of UI element that you want to be able to instantiate. Set up the UI element the way you want it to look in the **Scene**, and then drag the element into the **Project View** to make it into a **prefab**.

For example, a prefab for a button could be a Game Object with a Image **component** and a Button **component**, and a child Game Object with a Text **component**. Your setup might be different depending on your needs.

You might wonder why we don't have a API methods to create the various types of controls, including visuals and everything. The reason is that there are an infinite number of way e.g. a button could be setup. Does it use an image, text, or both? Maybe even multiple images? What is the text font, color, font size, and alignment? What sprite or **sprites** should the image use? By letting you make a prefab and instantiate that, you can set it up exactly the way you want. And if you later want to change the look and feel of your UI you can just change the prefab and then it will be reflected in your UI, including the dynamically created UI.

Instantiating the UI element

Prefabs of UI elements are instantiated as normal using the Instantiate method. When setting the parent of the instantiated UI element, it's recommended to do it using the Transform.SetParent method with the worldPositionStays parameter set to false.

Positioning the UI element

A UI Element is normally positioned using its Rect Transform. If the UI Element is a child of a Layout Group it will be automatically positioned and the positioning step can be skipped.

When positioning a Rect Transform it's useful to first determine it has or should have any stretching behavior or not. Stretching behavior happens when the anchorMin and anchorMax properties are not identical.

For a non-stretching Rect Transform, the position is set most easily by setting the anchoredPosition and the sizeDelta properties. The anchoredPosition specifies the position of the pivot relative to the anchors. The sizeDelta is just the same as the size when there's no stretching.

For a stretching Rect Transform, it can be simpler to set the position using the offsetMin and offsetMax properties. The offsetMin property specifies the corner of the lower left corner of the rect relative to the lower left anchor. The offsetMax property specifies the corner of the upper right corner of the rect relative to the upper right anchor.

Customizing the UI Element

If you are instantiating multiple UI elements dynamically, it's unlikely that you'll want them all to look the same and do the same. Whether it's buttons in a menu, items in an inventory, or something else, you'll likely want the individual items to have different text or images and to do different things when interacted with.

This is done by getting the various components and changing their properties. See the scripting reference for the Image and Text components, and for how to work with UnityEvents from scripting.

Creating Screen Transitions

Leave feedback

The need to transition between multiple **UI** screens is fairly common. In this page we will explore a simple way to create and manage those transitions using animation and **State Machines** to drive and control each screen.

Overview

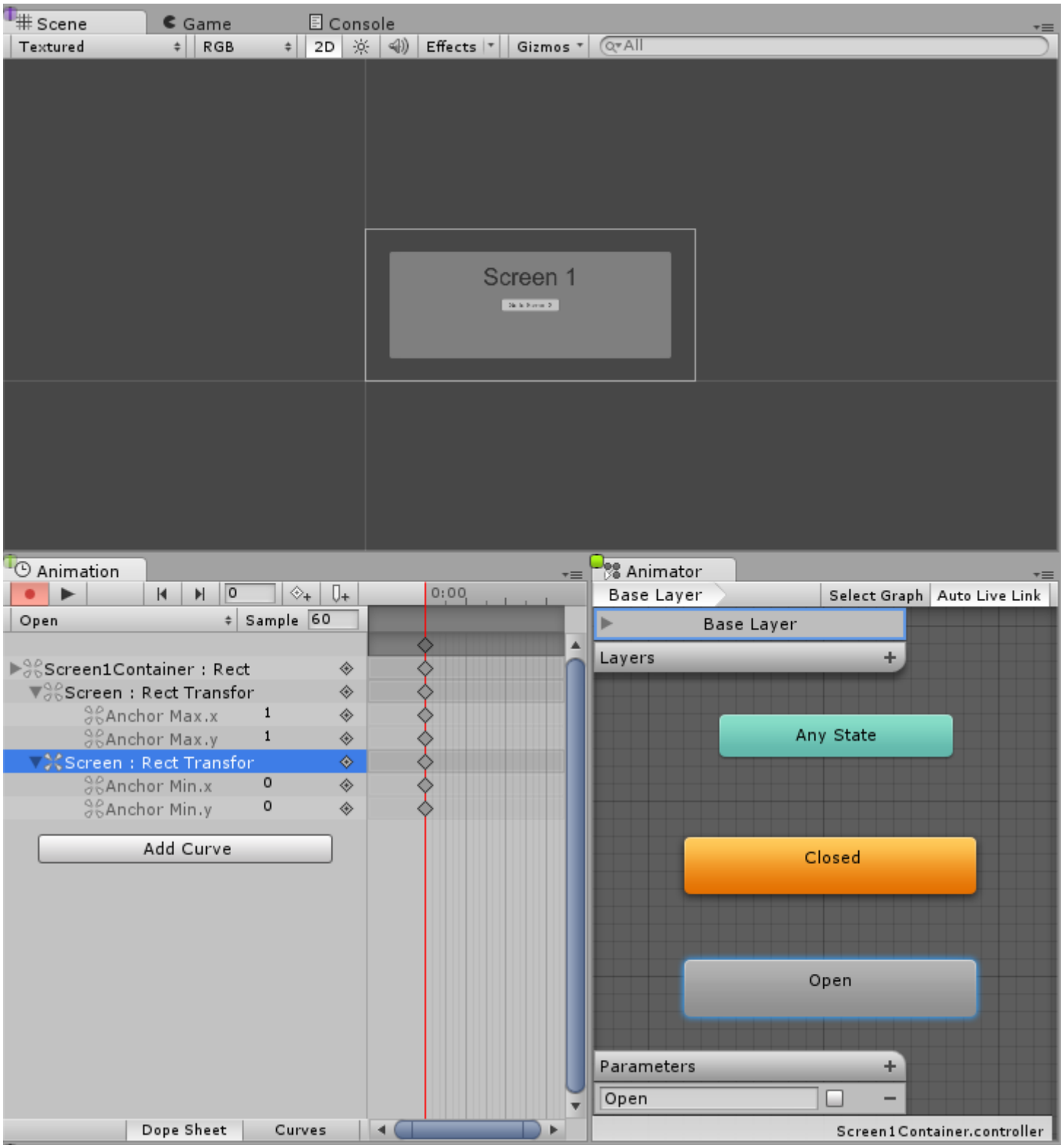
The high-level idea is that each of our screens will have an Animator Controller with two states (Open and Closed) and a boolean Parameter (Open). To transition between screens you will only need to close the currently open Screen and open the desired one. To make this process easier we will create a small Class ScreenManager that will keep track and take care of closing any already open Screen for us. The button that triggers the transition will only have to ask the ScreenManager to open the desired screen.

Thinking about Navigation

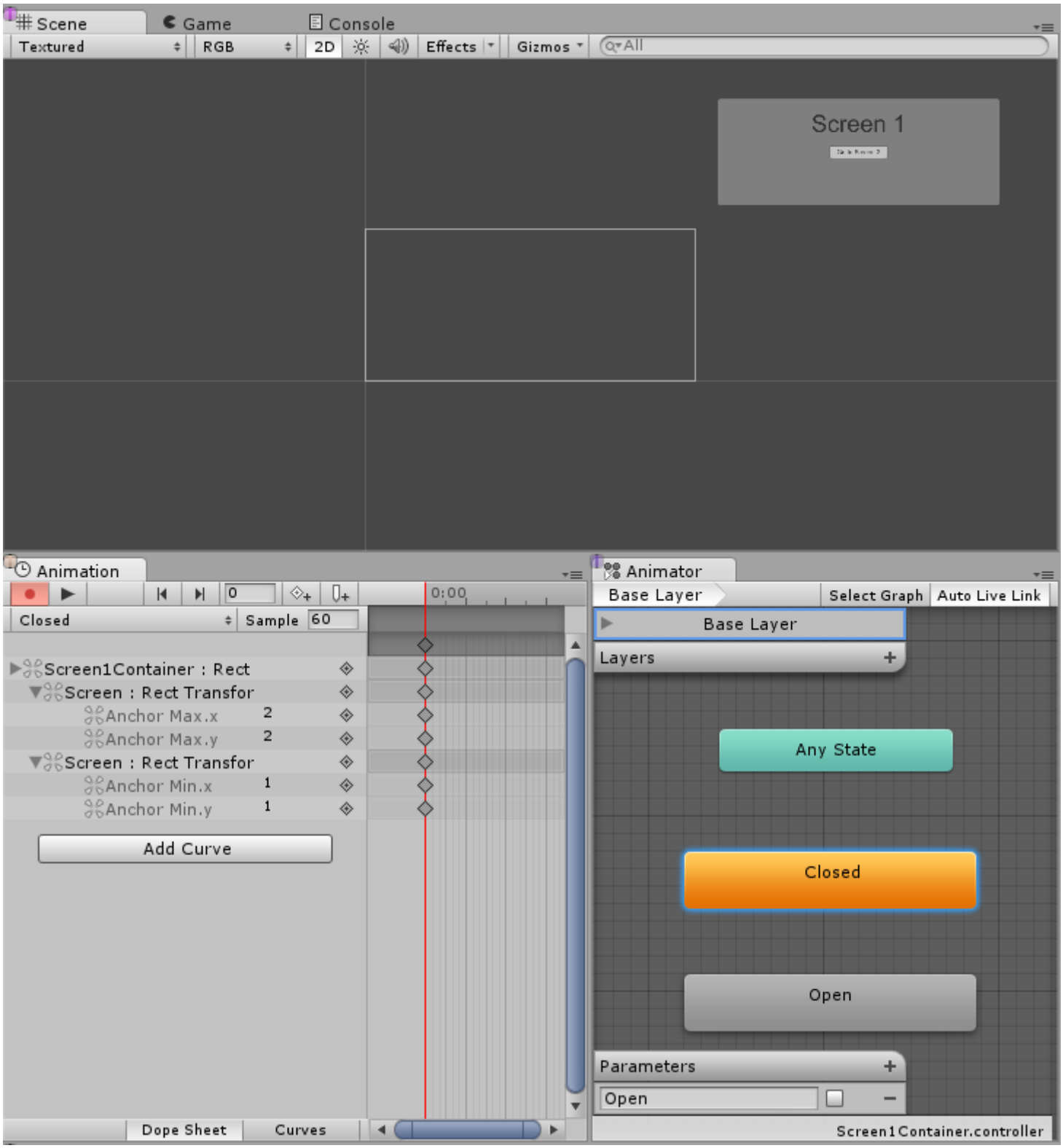
If you plan to support controller/keyboard navigation of UI elements, then it's important to have a few things in mind. It's important to avoid having Selectable elements outside the screen since that would enable players to select offscreen elements, we can do that by deactivating any off-screen hierarchy. We also need to make sure when a new screen is shown we set a element from it as selected, otherwise the player would not be able to navigate to the new screen. We will take care of all that in the ScreenManager class below.

Setting up the Animator Controller

Let's take a look at the most common and minimal setup for the Animation Controller to do a Screen transition. The controller will need a boolean parameter (Open) and two states (Open and Closed), each state should have an animation with only one **keyframe**, this way we let the State Machine do the transition blending for us.

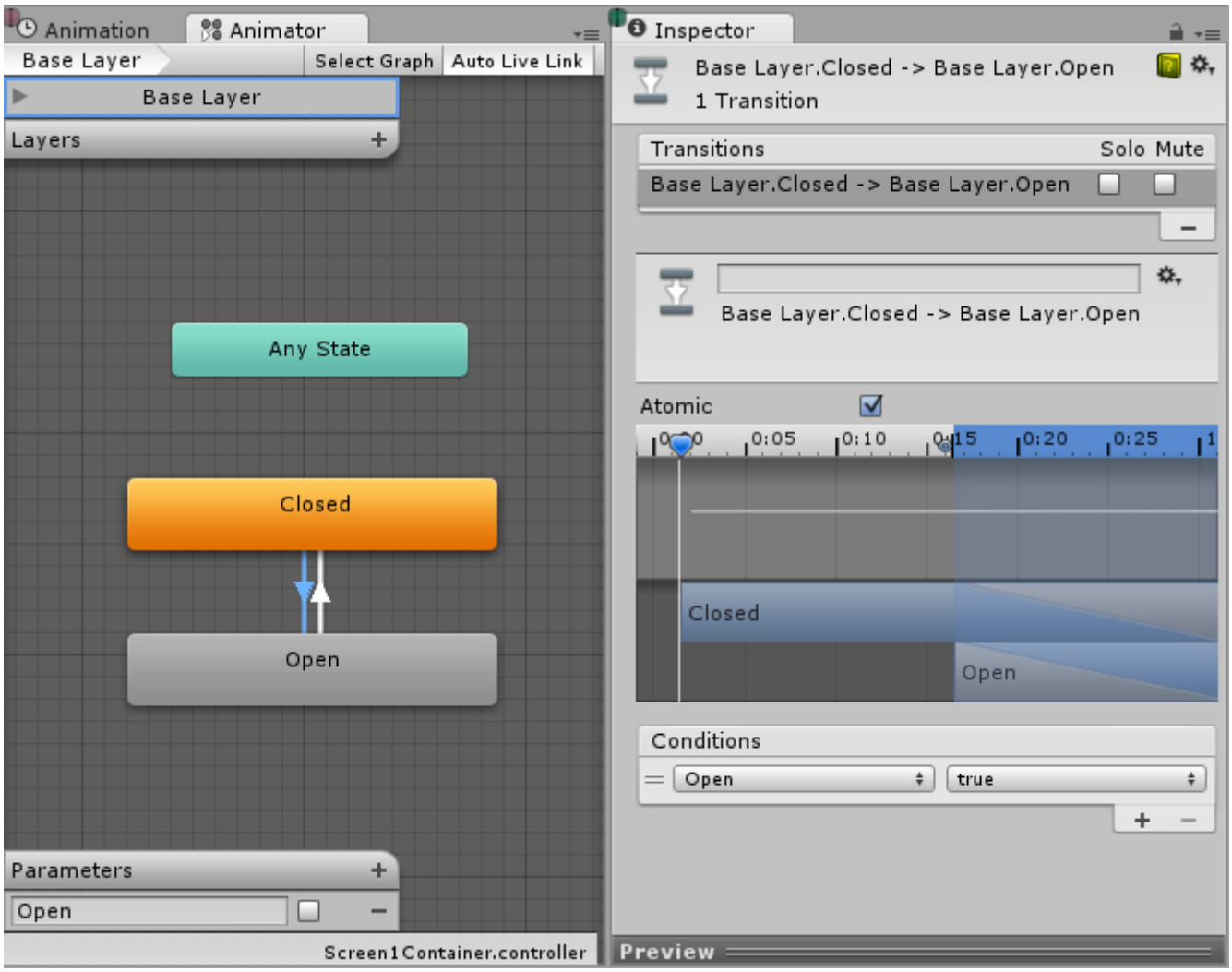


The Open state and animation

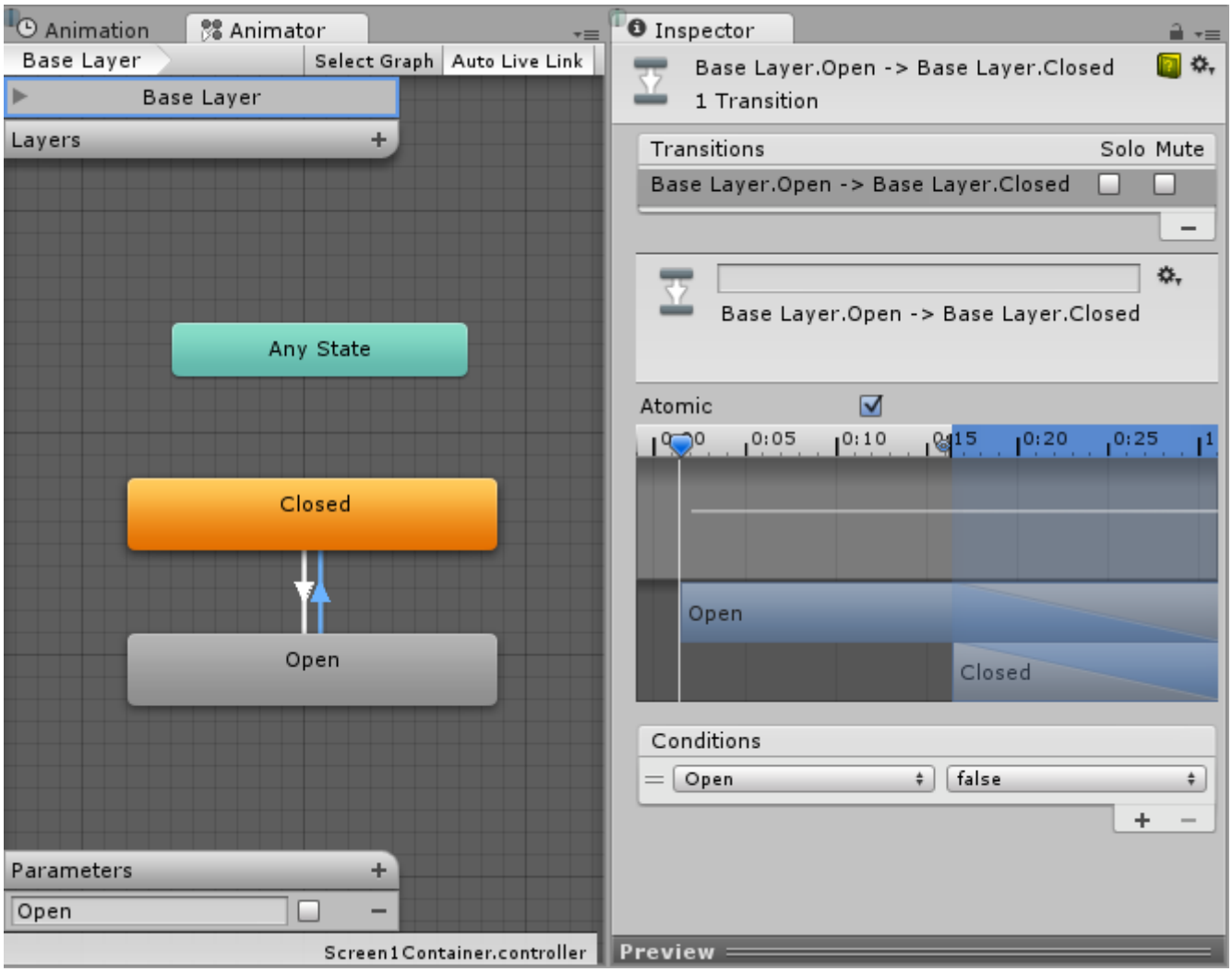


The Closed state and animation

Now we need to create the transition between both states, let's start with the transition from Open to Closed and let's set the condition properly, we want to go from Open to Closed when the parameter Open is set to false. Now we create the transition from Closed to Open and set the condition to go from Closed to Open when the parameter Open is true.



The Transition from Closed to Open



The Transition from Open to Closed

Managing the screens

With all the above set up, the only thing missing is for us to set the parameter Open to true on the screens Animator we want to transition to and Open to false on the currently open screens Animator. To do that, we will create a small script:

```
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.EventSystems;
using System.Collections;
using System.Collections.Generic;

public class ScreenManager : MonoBehaviour {

    //Screen to open automatically at the start of the Scene
    public Animator initiallyOpen;

    //Currently Open Screen
    private Animator m_Open;
```

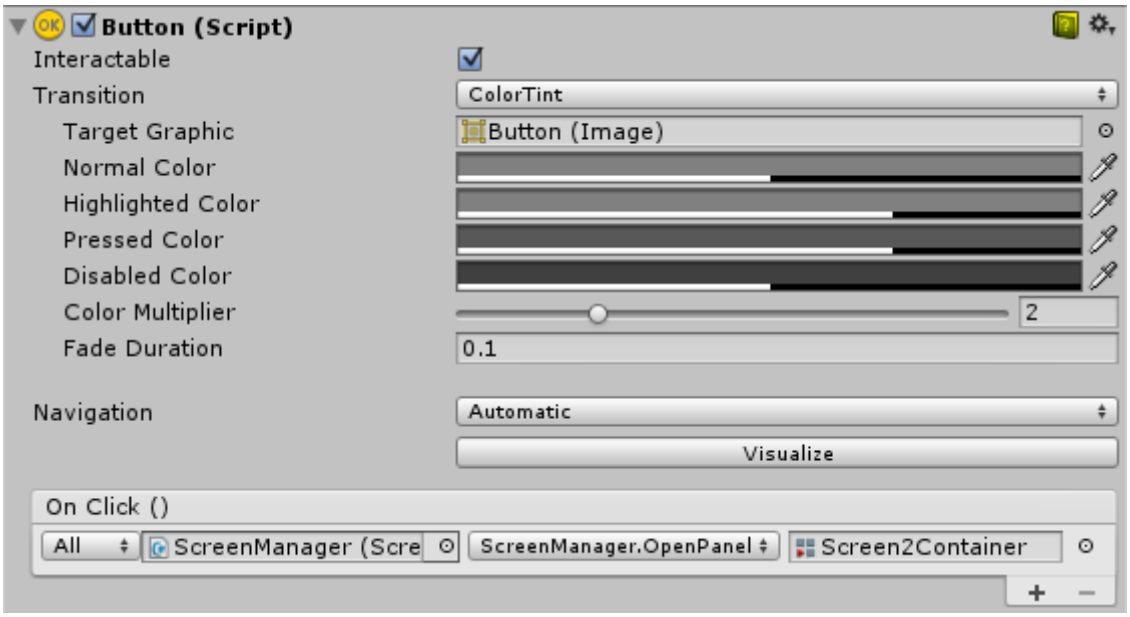
```
    //Hash of the parameter we use to control the transitions.
    private int m_OpenParameterId;

    //The GameObject Selected before we opened the current Screen.
    //Used when closing a Screen, so we can go back to the button that opened it
    private GameObject m_PreviouslySelected;

    //Animator State and Transition names we need to check against.
    const string k_OpenTransitionName = "Open";
    const string k_ClosedStateName = "Closed";

    public void OnEnable()
    {
        //We cache the Hash to the "Open" Parameter, so we can feed to Animator.
        m_OpenParameterId = Animator.StringToHash (k_OpenTransitionName);

        //If set, open the initial Screen now.
        if (initiallyOpen == null)
            return;
        OpenPanel(initiallyOpen);
    }

    //Closes the currently open panel and opens the provided one.
    //It also takes care of handling the navigation, setting the new Selected el
    public void OpenPanel (Animator anim)
    {
        if (m_Open == anim)
            return;

        //Activate the new Screen hierarchy so we can animate it.
        anim.gameObject.SetActive(true);
        //Save the currently selected button that was used to open this Screen.
        var newPreviouslySelected = EventSystem.current.currentSelectedGameObjec
        //Move the Screen to front.
        anim.transform.SetAsLastSibling();

        CloseCurrent();

        m_PreviouslySelected = newPreviouslySelected;

        //Set the new Screen as then open one.
        m_Open = anim;
        //Start the open animation
        m_Open.SetBool(m_OpenParameterId, true);

        //Set an element in the new screen as the new Selected one.
```

```
        GameObject go = FindFirstEnabledSelectable(anim.gameObject);
        SetSelected(go);
    }

    //Finds the first Selectable element in the providade hierarchy.
    static GameObject FindFirstEnabledSelectable (GameObject gameObject)
    {
        GameObject go = null;
        var selectables = gameObject.GetComponentsInChildren<Selectable> (true);
        foreach (var selectable in selectables) {
            if (selectable.IsActive () && selectable.IsInteractable ()) {
                go = selectable.gameObject;
                break;
            }
        }
        return go;
    }

    //Closes the currently open Screen
    //It also takes care of navigation.
    //Reverting selection to the Selectable used before opening the current scre
    public void CloseCurrent()
    {
        if (m_Open == null)
            return;

        //Start the close animation.
        m_Open.SetBool(m_OpenParameterId, false);

        //Reverting selection to the Selectable used before opening the current
        SetSelected(m_PreviouslySelected);
        //Start Coroutine to disable the hierarchy when closing animation finish
        StartCoroutine(DisablePanelDeleyed(m_Open));
        //No screen open.
        m_Open = null;
    }

    //Coroutine that will detect when the Closing animation is finished and it w
    //hierarchy.
    IEnumerator DisablePanelDeleyed(Animator anim)
    {
        bool closedStateReached = false;
        bool wantToClose = true;
        while (!closedStateReached && wantToClose)
        {
            if (!anim.IsInTransition(0))
```

```
                closedStateReached = anim.GetCurrentAnimatorStateInfo(0).IsName(

            wantToClose = !anim.GetBool(m_OpenParameterId);

            yield return new WaitForEndOfFrame();
        }

        if (wantToClose)
            anim.gameObject.SetActive(false);
    }

    //Make the provided GameObject selected
    //When using the mouse/touch we actually want to set it as the previously se
    //set nothing as selected for now.
    private void SetSelected(GameObject go)
    {
        //Select the GameObject.
        EventSystem.current.SetSelectedGameObject(go);

        //If we are using the keyboard right now, that's all we need to do.
        var standaloneInputModule = EventSystem.current.currentInputModule as St
        if (standaloneInputModule != null)
            return;

        //Since we are using a pointer device, we don't want anything selected.
        //But if the user switches to the keyboard, we want to start the navigat
        //So here we set the current Selected to null, so the provided gameObjec
        EventSystem.current.SetSelectedGameObject(null);
    }
}
```

Let's hook up this script, we do this by creating a new **GameObject**, we can rename it "ScreenManager" for instance, and add the **component** above to it. You can assign an initial screen to it, this screen will be open at the start of your **scene**.

Now for the final part, let's make the UI buttons work. Select the button that should trigger the screen transition and add a new action under the **On Click ()** list in the Inspector. Drag the ScreenManager GameObject we just created to the ObjectField, on the dropdown select **ScreenManager->OpenPanel (Animator)** and drag and drop the panel you want to open when the user clicks the button to the las ObjectField.



Button Inspector

Notes

This technique only requires each screen to have an AnimatorController with an Open parameter and a Closed state to work - it doesn't matter how your screen or State Machine are constructed. This technique also works well with nested screens, meaning you only need one ScreenManager for each nested level.

The State Machine we set up above has the default state of Closed, so all of the screens that use this controller start as closed. The ScreenManager provides an initiallyOpen property so you can specify which screen is shown first.

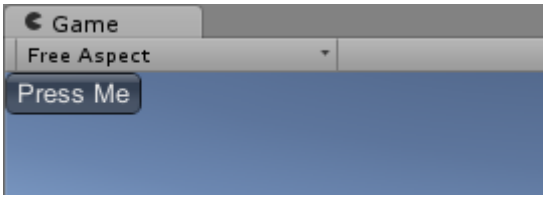
Immediate Mode GUI (IMGUI)

Leave feedback

The “Immediate Mode” GUI system (also known as IMGUI) is an entirely separate feature to Unity’s main GameObject-based **UI** System. IMGUI is a code-driven GUI system, and is mainly intended as a tool for programmers. It is driven by calls to the **OnGUI** function on any script which implements it. For example, this code:

```
void OnGUI() {
    if (GUILayout.Button("Press Me"))
        Debug.Log("Hello!");
}
```

Would result in a button displayed like so:



The result of the above code example

The Immediate Mode GUI system is commonly used for:

- Creating in-game debugging displays and tools.
- Creating custom **inspectors** for script **components**.
- Creating new editor windows and tools to extend Unity itself.

The IMGUI system is not generally intended to be used for normal in-game user interfaces that players might use and interact with. For that you should use Unity’s main GameObject-based UI system, which offers a GameObject-based approach for editing and positioning UI elements, and has far better tools to work with the visual design and layout of the UI.

“Immediate Mode” refers to the way the IMGUI is created and drawn. To create IMGUI elements, you must write code that goes into a special function named OnGUI. The code to display the interface is executed every frame, and drawn to the screen. There are no persistent **gameobjects** other than the object to which your OnGUI code is attached, or other types of objects in the hierarchy related to the visual elements that are drawn.

IMGUI allows you to create a wide variety of functional GUIs using code. Rather than creating GameObjects, manually positioning them, and then writing a script that handles its functionality, you can do everything at once with just a few lines of code. The code produces **GUI controls** that are drawn and handled with a single function call.

This section explains how to use IMGUI both in your game and in extensions to the Unity editor.

IMGUI Basics

Leave feedback

This section will explain the bare necessities for scripting **Controls** with Unity's Immediate Mode GUI system (IMGUI).

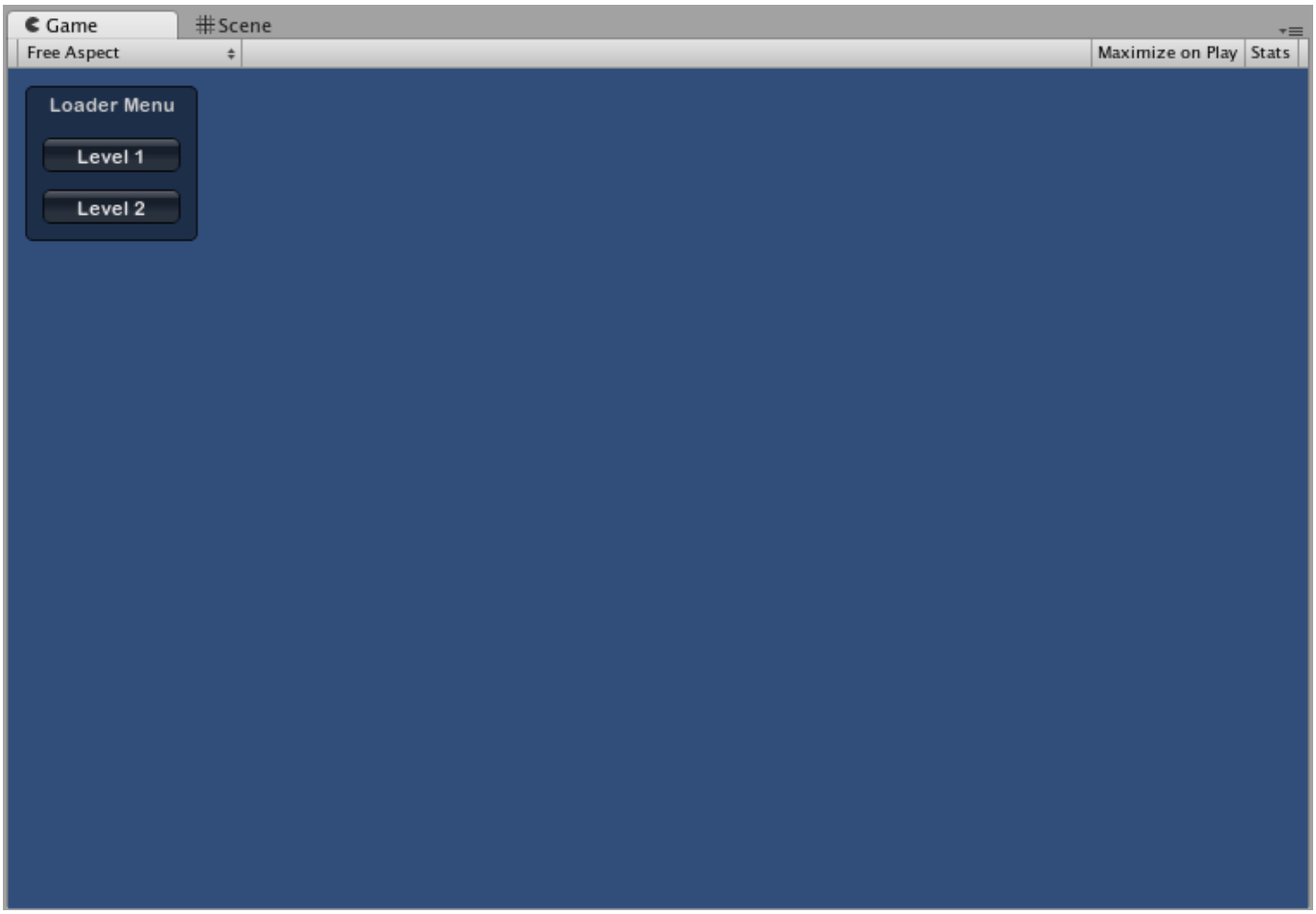
Making Controls with IMGUI

Unity's IMGUI controls make use of a special function called **OnGUI()**. The **OnGUI()** function gets called every frame as long as the containing script is enabled - just like the **Update()** function.

IMGUI controls themselves are very simple in structure. This structure is evident in the following example.

```
/* Example level loader */
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour {

    void OnGUI ()
    {
        // Make a background box
        GUI.Box(new Rect(10,10,100,90), "Loader Menu");

        // Make the first button. If it is pressed, Application.Loadlevel (1) will be exec
        if(GUI.Button(new Rect(20,40,80,20), "Level 1"))
        {
            Application.LoadLevel(1);
        }

        // Make the second button.
        if(GUI.Button(new Rect(20,70,80,20), "Level 2"))
        {
            Application.LoadLevel(2);
        }
    }
}
```

This example is a complete, functional level loader. If you copy/paste this script and attach it a **GameObject**, you'll see the following menu appear in when you enter **Play Mode**:



The Loader Menu created by the example code

Let's take a look at the details of the example code:

The first GUI line, **GUI.Box (Rect (10,10,100,90), "Loader Menu");** displays a **Box** Control with the header text "Loader Menu". It follows the typical GUI Control declaration scheme which we'll explore momentarily.

The next GUI line is a **Button** Control declaration. Notice that it is slightly different from the Box Control declaration. Specifically, the entire Button declaration is placed inside an **if** statement. When the game is running and the Button is clicked, this **if** statement returns true and any code inside the **if** block is executed.

Since the **OnGUI()** code gets called every frame, you don't need to explicitly create or destroy GUI controls. The line that declares the Control is the same one that creates it. If you need to display Controls at specific times, you can use any kind of scripting logic to do so.

```
/* Flashing button example */
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour
{

    void OnGUI ()
    {
        if (Time.time % 2 < 1)
        {
            if (GUI.Button (new Rect (10,10,200,20), "Meet the flashing button"))
```

```
            {
                print ("You clicked me!");
            }
        }
    }
}
```

Here, **GUI.Button()** only gets called every other second, so the button will appear and disappear. Naturally, the user can only click it when the button is visible.

As you can see, you can use any desired logic to control when GUI Controls are displayed and functional. Now we will explore the details of each Control's declaration.

Anatomy of a Control

There are three key pieces of information required when declaring a GUI Control:

Type (**Position**, **Content**)

Observe that this structure is a function with two arguments. We'll explore the details of this structure now.

Type

Type is the **Control Type**, and is declared by calling a function in Unity's GUI class or the GUILayout class, which is discussed at length in the Layout Modes section of the Guide. For example, **GUI.Label()** will create a non-interactive label. All the different control types are explained later, in the Controls section of the Guide.

Position

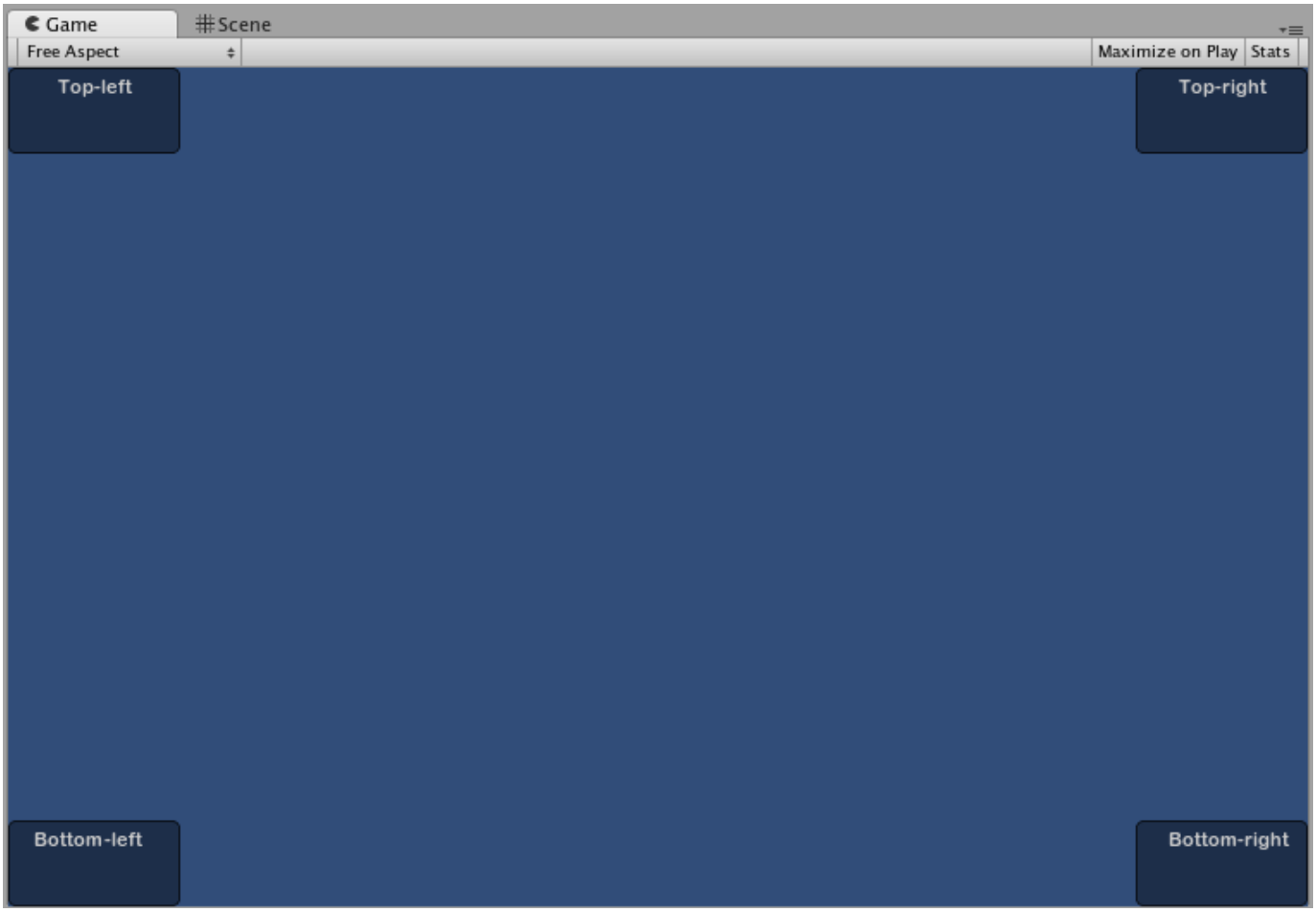
The **Position** is the first argument in any **GUI** Control function. The argument itself is provided with a **Rect()** function. **Rect()** defines four properties: **left-most position**, **top-most position**, **total width**, **total height**. All of these values are provided in **integers**, which correspond to pixel values. All UnityGUI controls work in **Screen Space**, which is the resolution of the published player in **pixels**.

The coordinate system is top-left based. **Rect(10, 20, 300, 100)** defines a Rectangle that starts at coordinates: 10,20 and ends at coordinates 310,120. It is worth repeating that the second pair of values in **Rect()** are total width and height, not the coordinates where the controls end. This is why the example mentioned above ends at 310,120 and not 300,100.

You can use the **Screen.width** and **Screen.height** properties to get the total dimensions of the screen space available in the player. The following example may help clarify how this is done:

```
/* Screen.width & Screen.height example */
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour
{

    void OnGUI()
    {
        GUI.Box (new Rect (0,0,100,50), "Top-left");
        GUI.Box (new Rect (Screen.width - 100,0,100,50), "Top-right");
```

```
        GUI.Box (new Rect (0,Screen.height - 50,100,50), "Bottom-left");
        GUI.Box (new Rect (Screen.width - 100,Screen.height - 50,100,50), "Bottom-right");
    }

}
```



The Boxes positioned by the above example

Content

The second argument for a GUI Control is the actual content to be displayed with the Control. Most often you will want to display some text or an image on your Control. To display text, pass a string as the Content argument like this:

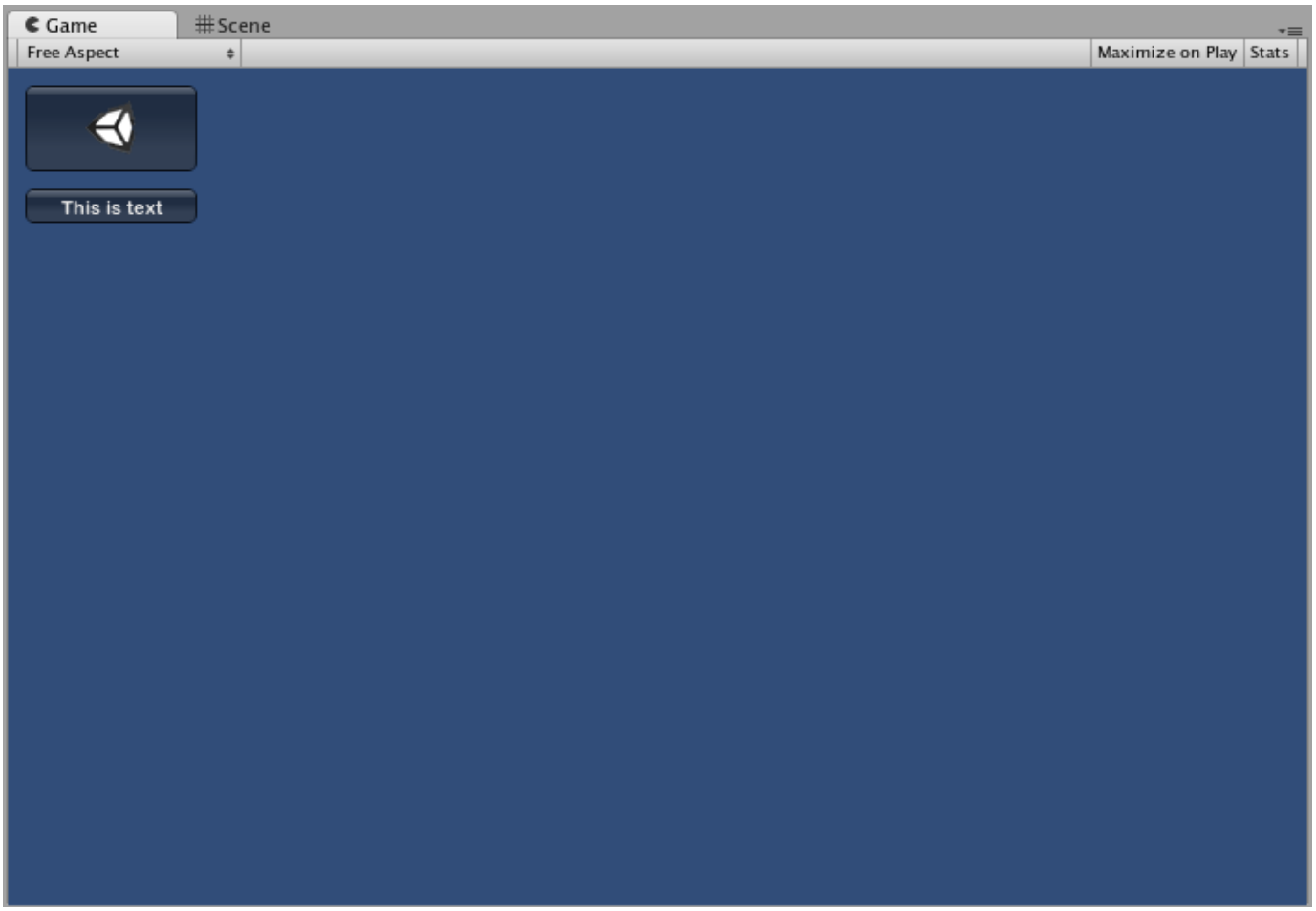
```
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour
{
    void OnGUI ()
    {
        GUI.Label (new Rect (0,0,100,50), "This is the text string for a Label Control");
    }
}
```

To display an image, declare a **Texture2D** public variable, and pass the variable name as the content argument like this:

```
/* Texture2D Content example */
public Texture2D controlTexture;
  ...

void OnGUI ()
{
    GUI.Label (new Rect (0,0,100,50), controlTexture);
}
```

Here is an example closer to a real-world scenario:

```
/* Button Content examples */
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour
{
    public Texture2D icon;

    void OnGUI ()
    {
        if (GUI.Button (new Rect (10,10, 100, 50), icon))
        {
            print ("you clicked the icon");
        }

        if (GUI.Button (new Rect (10,70, 100, 20), "This is text"))
        {
            print ("you clicked the text button");
        }
    }
}
```



The Buttons created by the above example

There is a third option which allows you to display images and text together in a GUI Control. You can provide a **GUIContent** object as the Content argument, and define the string and image to be displayed within the GUIContent.

```
/* Using GUIContent to display an image and a string */
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour
{
    public Texture2D icon;

    void OnGUI ()
    {
        GUI.Box (new Rect (10,10,100,50), new GUIContent("This is text", icon));
    }
}
```

You can also define a **Tooltip** in the GUIContent, and display it elsewhere in the GUI when the mouse hovers over it.

```
/* Using GUIContent to display a tooltip */
using UnityEngine;
```

```
using System.Collections;

public class GUITest : MonoBehaviour
{
    void OnGUI ()
    {
        // This line feeds "This is the tooltip" into GUI.tooltip
        GUI.Button (new Rect (10,10,100,20), new GUIContent ("Click me", "This is the tool

        // This line reads and displays the contents of GUI.tooltip
        GUI.Label (new Rect (10,40,100,20), GUI.tooltip);
    }
}
```

You can also use GUIContent to display a string, an icon, and a tooltip.

```
/* Using GUIContent to display an image, a string, and a tooltip */
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour
{
    public Texture2D icon;

    void OnGUI ()
    {
        GUI.Button (new Rect (10,10,100,20), new GUIContent ("Click me", icon, "This is th
        GUI.Label (new Rect (10,40,100,20), GUI.tooltip);
    }
}
```

The script reference page for the GUIContent constructor has some examples of its use.

Controls

Leave feedback

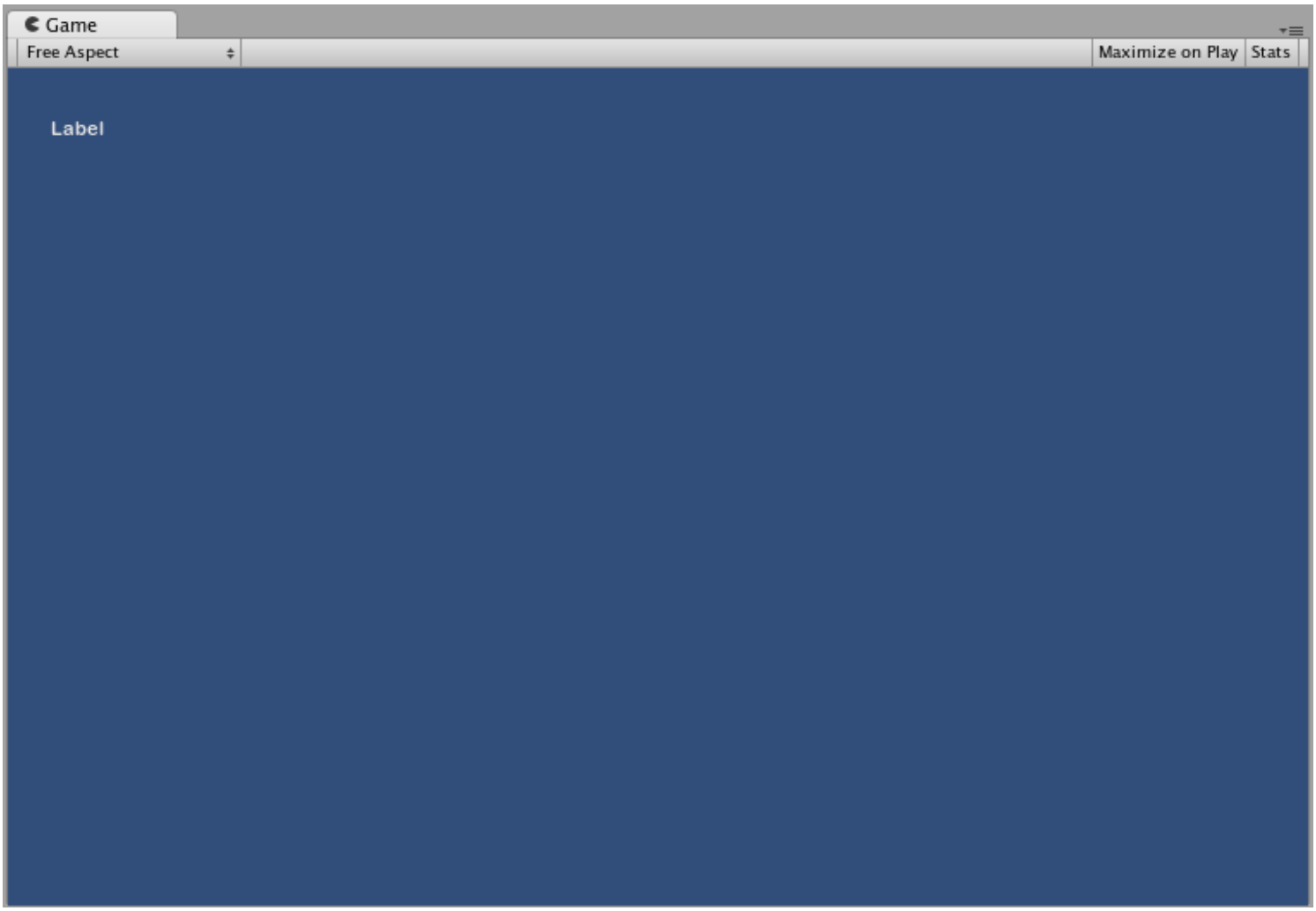
IMGUI Control Types

There are a number of different IMGUI **Controls** that you can create. This section lists all of the available display and interactive Controls. There are other IMGUI functions that affect layout of Controls, which are described in the Layout section of the Guide.

Label

The **Label** is non-interactive. It is for display only. It cannot be clicked or otherwise moved. It is best for displaying information only.

```
/* GUI.Label example */

using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour
{

    void OnGUI ()
    {
        GUI.Label (new Rect (25, 25, 100, 30), "Label");
    }

}
```



The Label created by the example code

Button

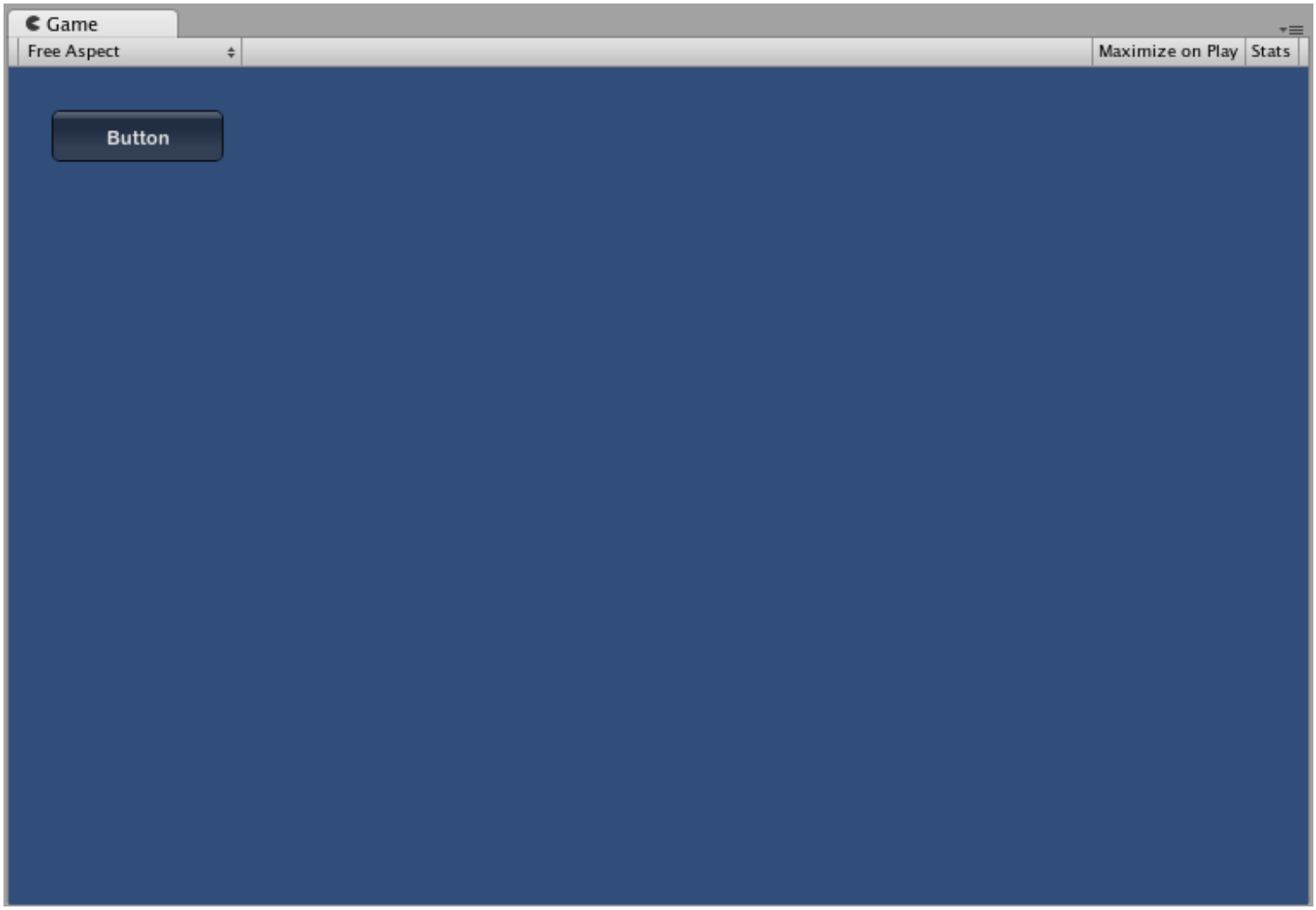
The **Button** is a typical interactive button. It will respond a single time when clicked, no matter how long the mouse remains depressed. The response occurs as soon as the mouse button is released.

Basic Usage

In UnityGUI, Buttons will return **true** when they are clicked. To execute some code when a Button is clicked, you wrap the the GUI.Button function in an **if** statement. Inside the **if** statement is the code that will be executed when the Button is clicked.

```
/* GUI.Button example */

using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour
{

    void OnGUI ()
    {
```

```
        if (GUI.Button (new Rect (25, 25, 100, 30), "Button"))
        {
            // This code is executed when the Button is clicked
        }
    }

}
```



The Button created by the example code

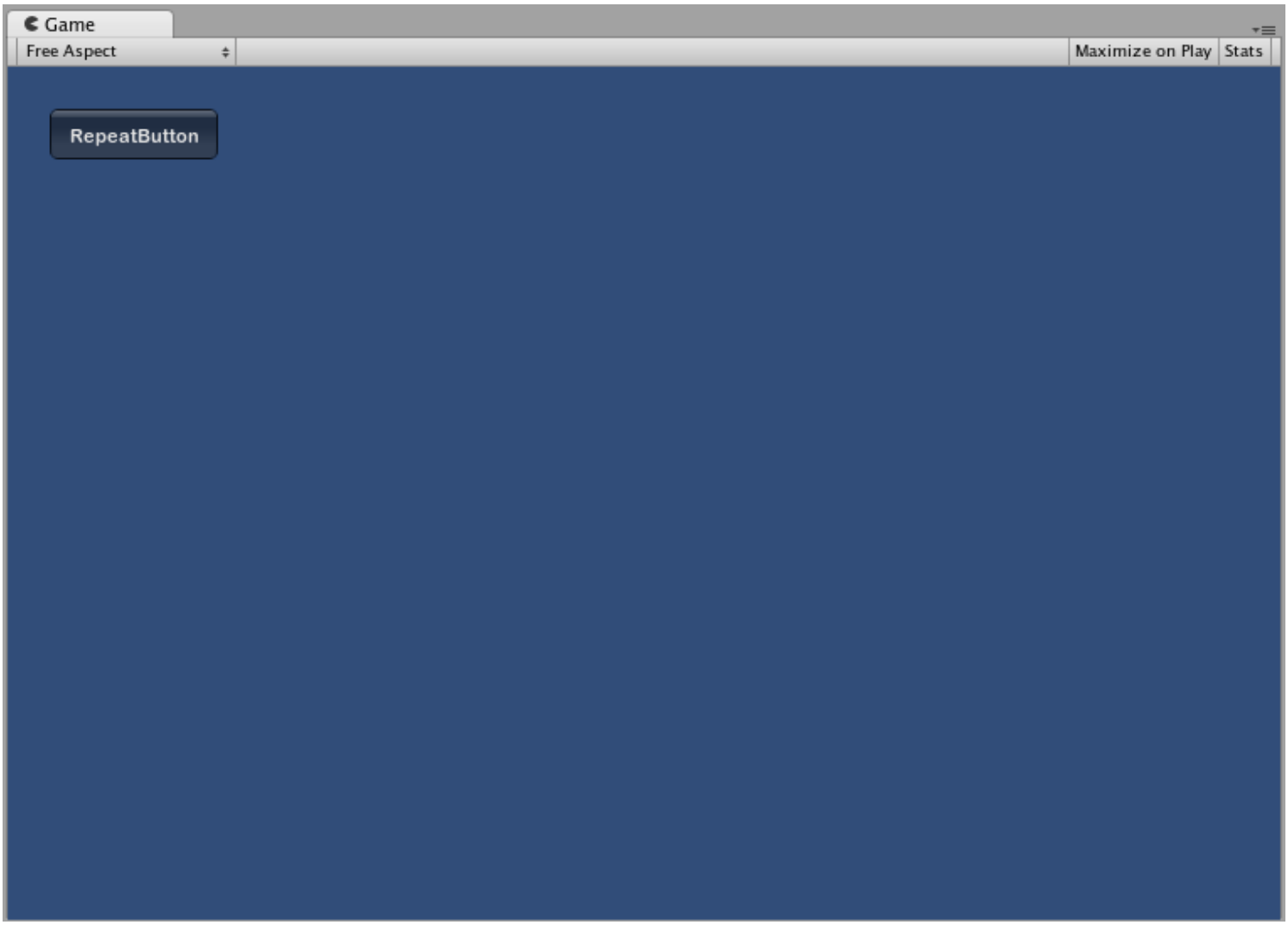
RepeatButton

RepeatButton is a variation of the regular **Button**. The difference is, **RepeatButton** will respond every frame that the mouse button remains depressed. This allows you to create click-and-hold functionality.

Basic Usage

In UnityGUI, RepeatButtons will return **true** for every frame that they are clicked. To execute some code while the Button is being clicked, you wrap the the GUI.RepeatButton function in an **if** statement. Inside the **if** statement is the code that will be executed while the RepeatButton remains clicked.

```
/* GUI.RepeatButton example */

using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour
{

    void OnGUI ()
    {
        if (GUI.RepeatButton (new Rect (25, 25, 100, 30), "RepeatButton"))
        {
            // This code is executed every frame that the RepeatButton remains c
        }
    }

}
```



The Repeat Button created by the example code

TextField

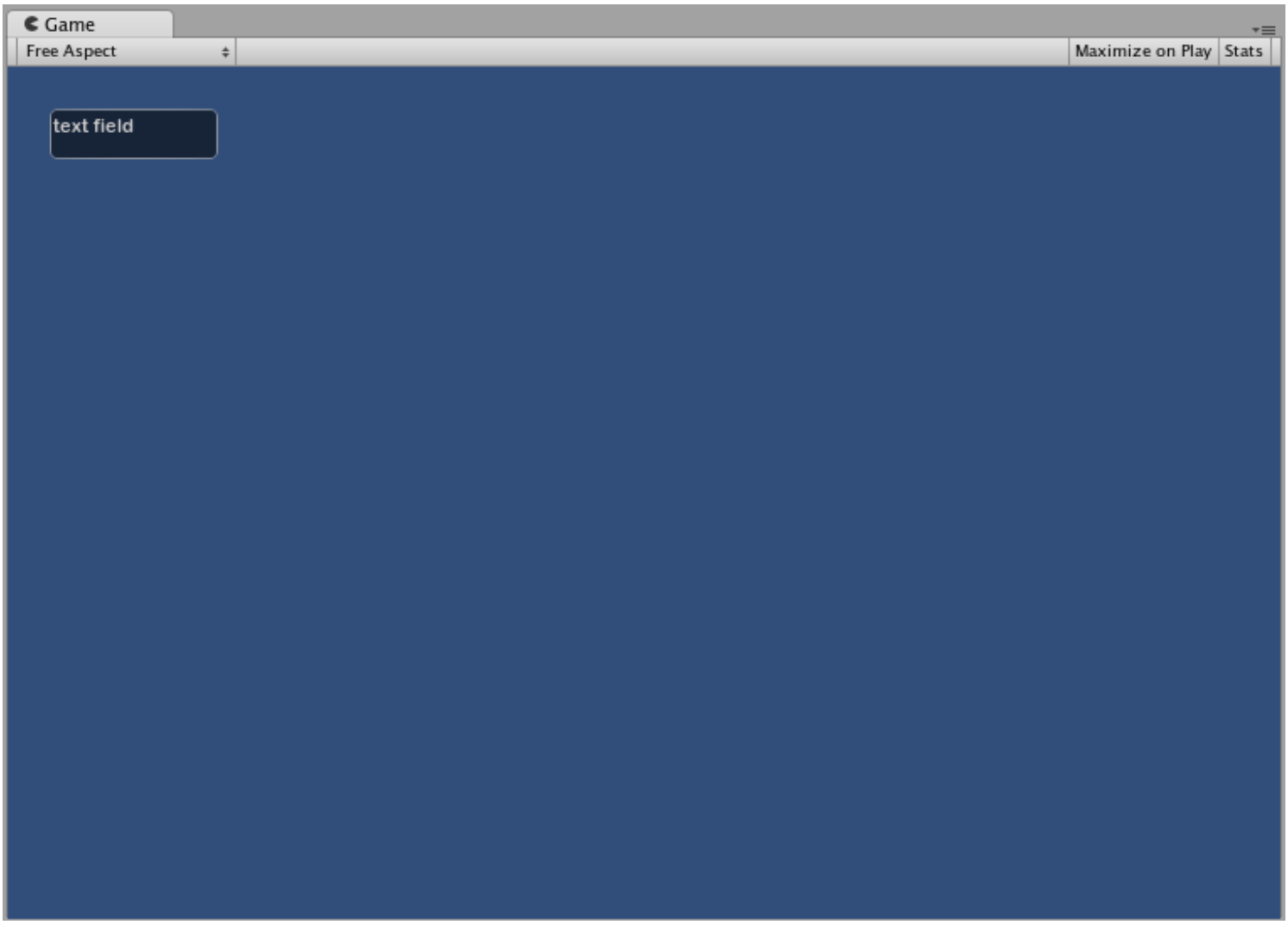
The **TextField** Control is an interactive, editable single-line field containing a text string.

Basic Usage

The TextField will always display a string. You must provide the string to be displayed in the TextField. When edits are made to the string, the TextField function will return the edited string.

```
/* GUI.TextField example */

using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour
{

    private string textFieldString = "text field";

    void OnGUI ()
```

```
    {
        textFieldString = GUI.TextField (new Rect (25, 25, 100, 30), textFieldSt
    }

}
```



The TextField created by the example code

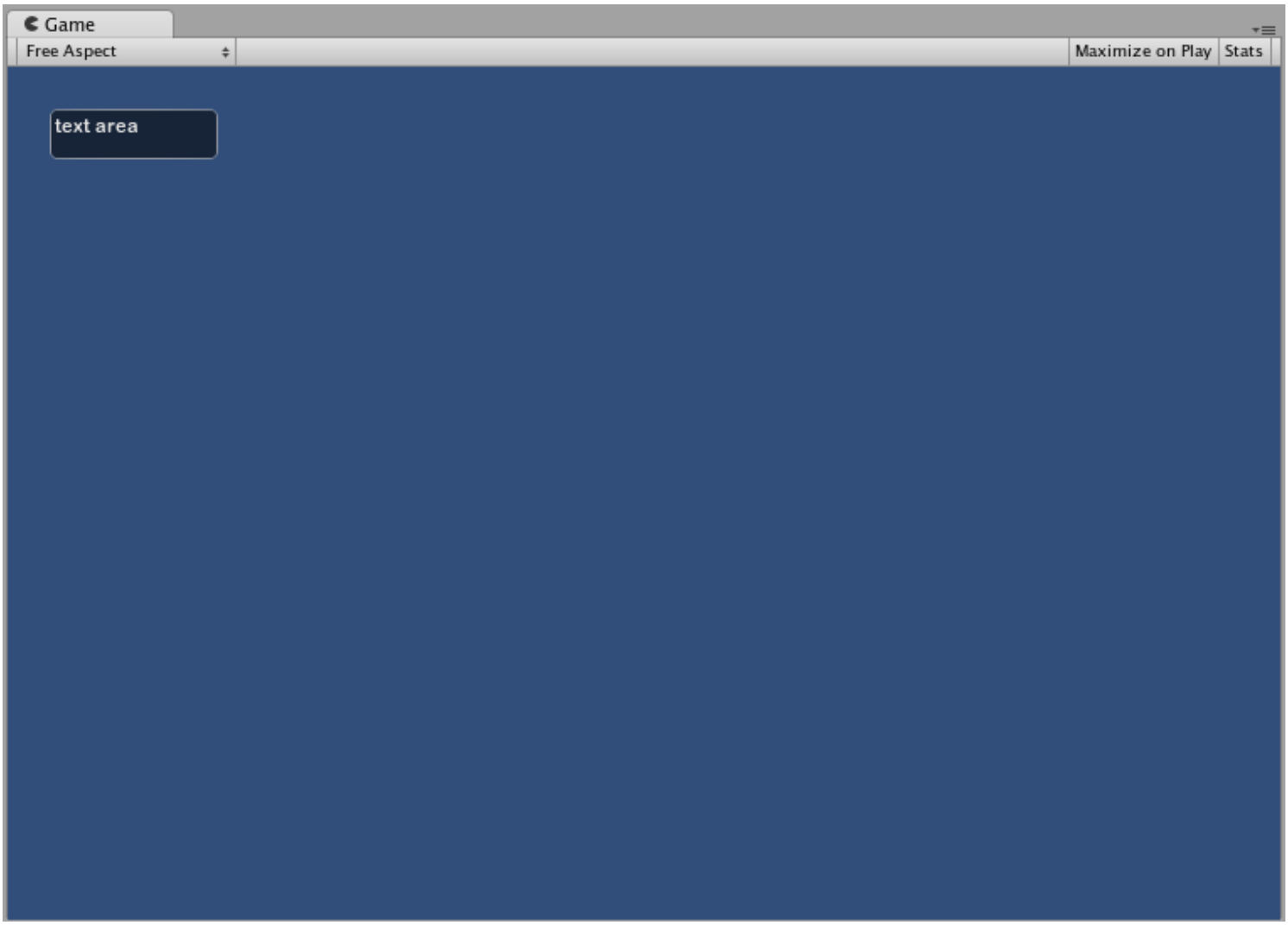
TextArea

The **TextArea** Control is an interactive, editable multi-line area containing a text string.

Basic Usage

The TextArea will always display a string. You must provide the string to be displayed in the TextArea. When edits are made to the string, the TextArea function will return the edited string.

```
/* GUI.TextArea example */

using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour
{

    private string textAreaString = "text area";

    void OnGUI ()
    {
        textAreaString = GUI.TextArea (new Rect (25, 25, 100, 30), textAreaStrin
    }

}
```



The TextArea created by the example code

Toggle

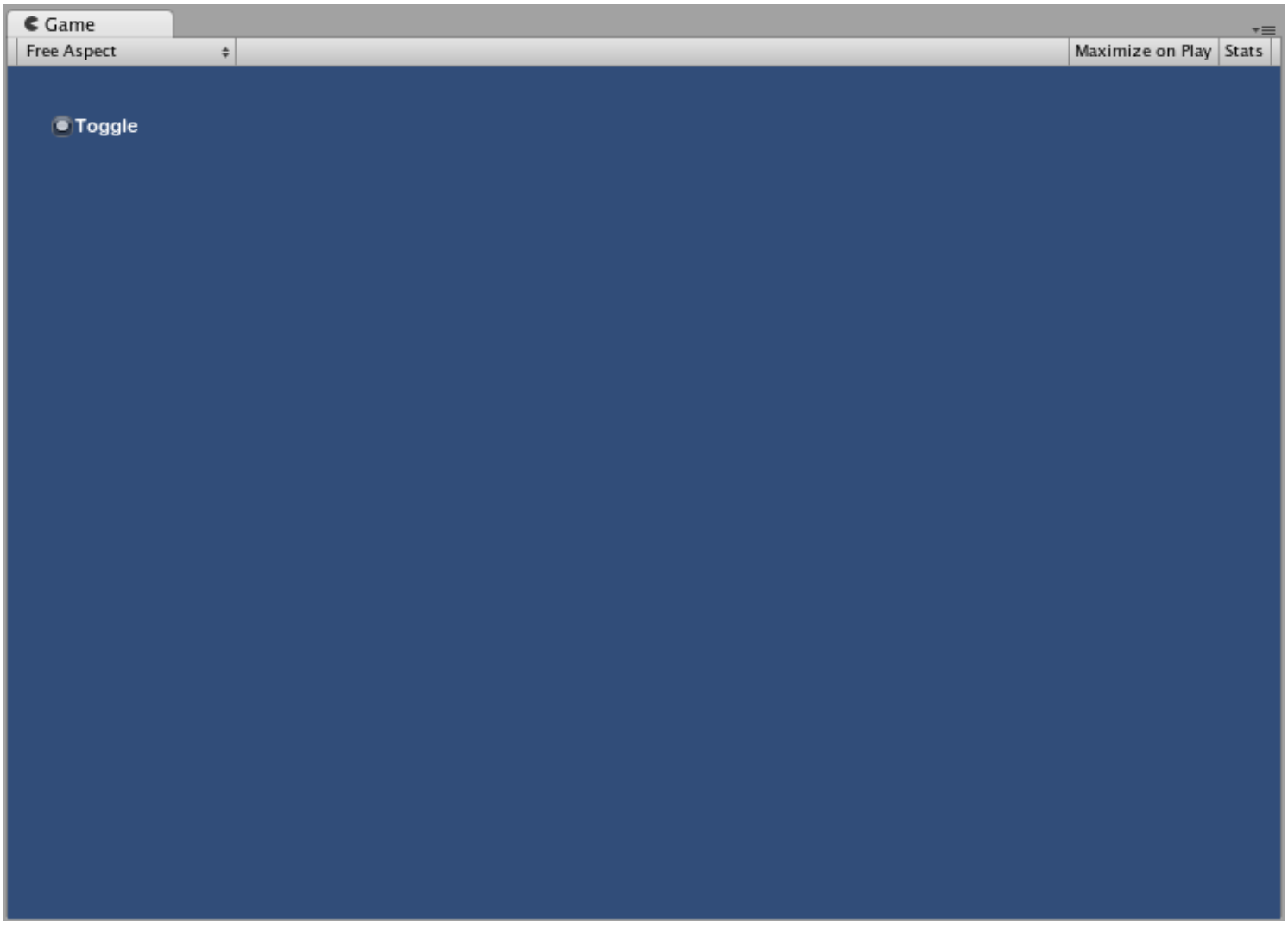
The **Toggle** Control creates a checkbox with a persistent on/off state. The user can change the state by clicking on it.

Basic Usage

The Toggle on/off state is represented by a true/false boolean. You must provide the boolean as a parameter to make the Toggle represent the actual state. The Toggle function will return a new boolean value if it is clicked. In order to capture this interactivity, you must assign the boolean to accept the return value of the Toggle function.

```
/* GUI.Toggle example */

using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour
{
    private bool toggleBool = true;
```

```
    void OnGUI ()
    {
        toggleBool = GUI.Toggle (new Rect (25, 25, 100, 30), toggleBool, "Toggle
    }
}
```



The Toggle created by the example code

Toolbar

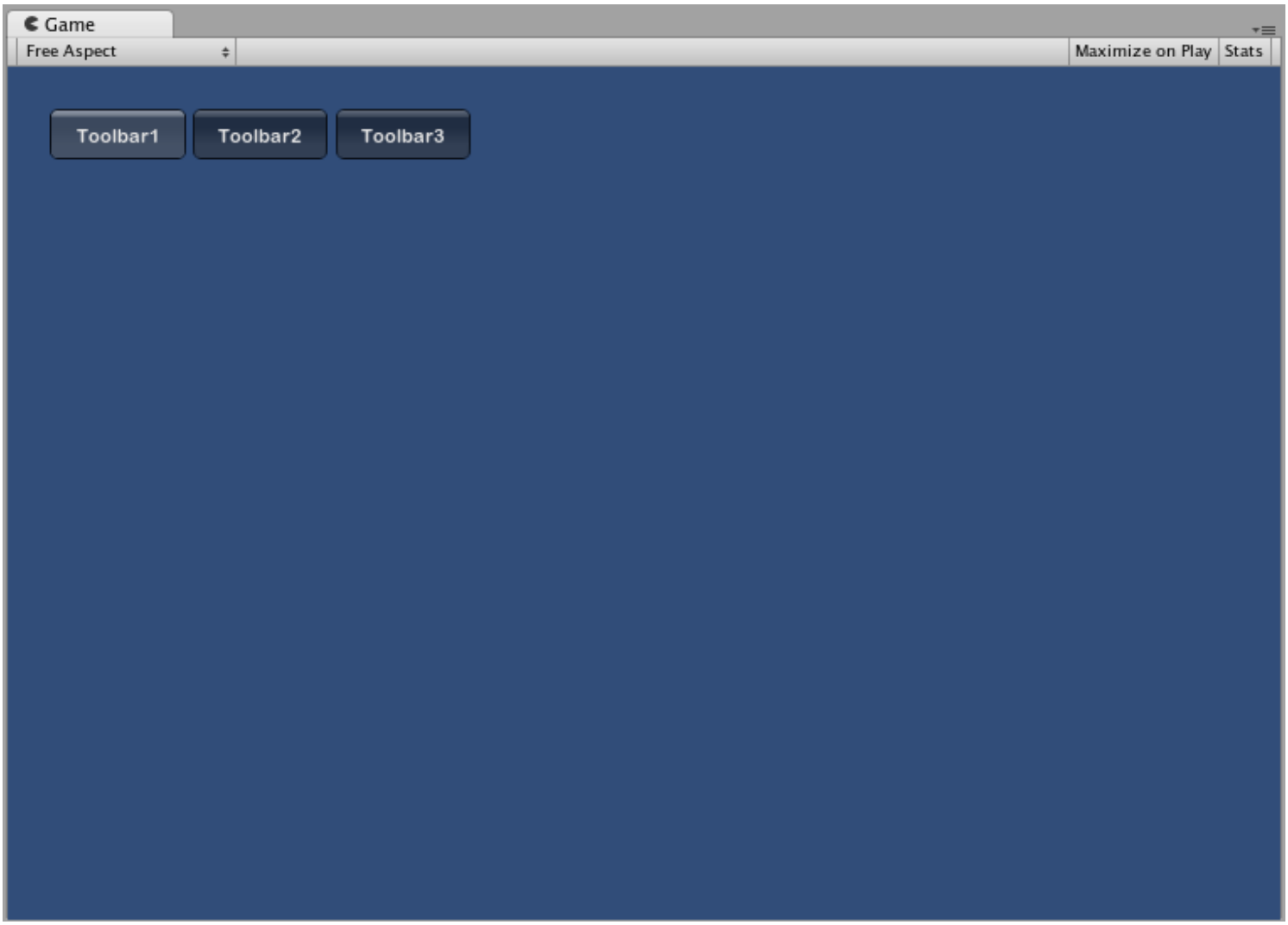
The **Toolbar** Control is essentially a row of **Buttons**. Only one of the Buttons on the Toolbar can be active at a time, and it will remain active until a different Button is clicked. This behavior emulates the behavior of a typical Toolbar. You can define an arbitrary number of Buttons on the Toolbar.

Basic Usage

The active Button in the Toolbar is tracked through an integer. You must provide the integer as an argument in the function. To make the Toolbar interactive, you must assign the integer to the return value of the function. The

number of elements in the content array that you provide will determine the number of Buttons that are shown in the Toolbar.

```
/* GUI.Toolbar example */

using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour
{

    private int toolbarInt = 0;
    private string[] toolbarStrings = {"Toolbar1", "Toolbar2", "Toolbar3"};

    void OnGUI ()
    {
        toolbarInt = GUI.Toolbar (new Rect (25, 25, 250, 30), toolbarInt, toolba
    }

}
```



The Toolbar created by the example code

SelectionGrid

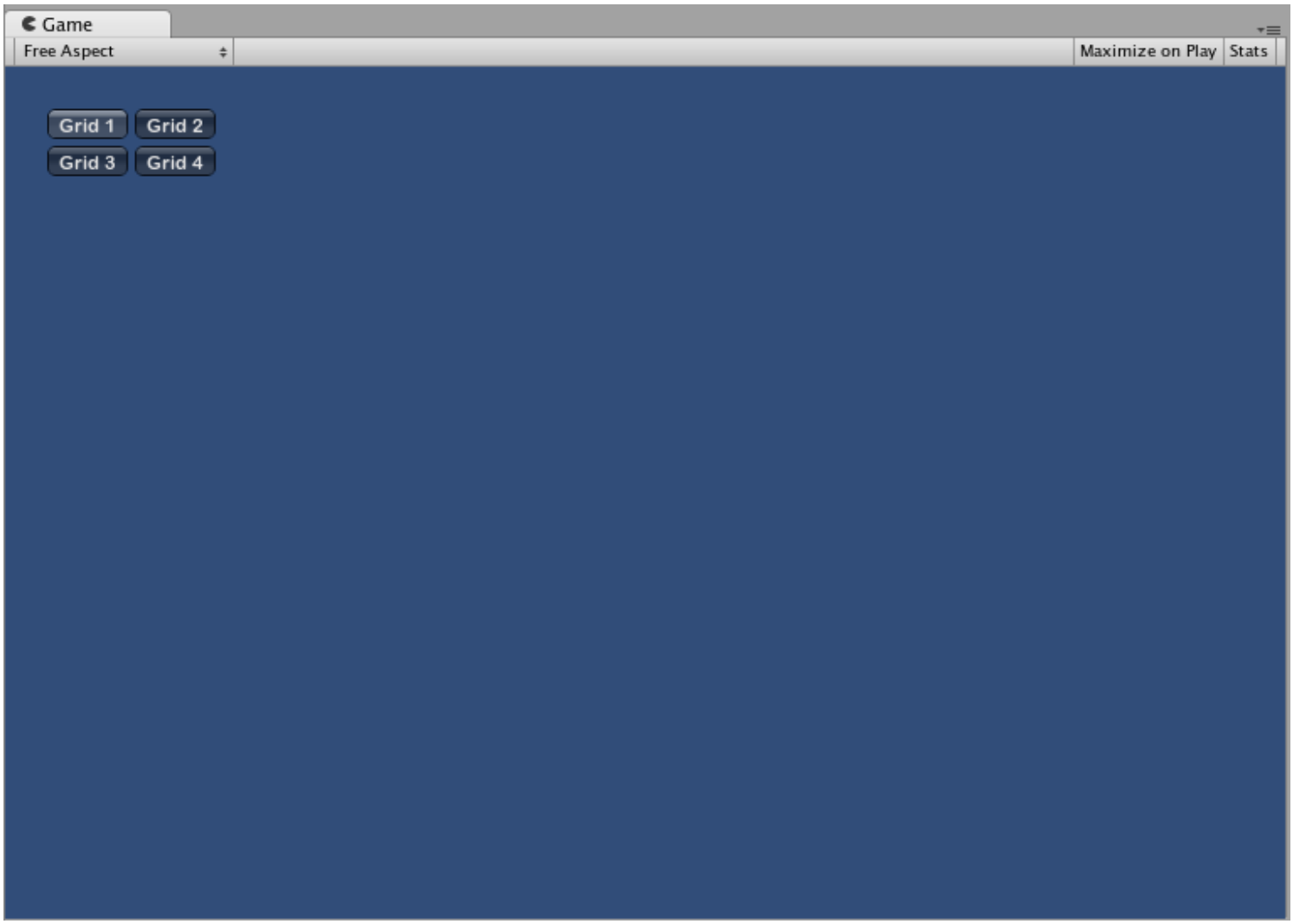
The **SelectionGrid** Control is a multi-row **Toolbar**. You can determine the number of columns and rows in the grid. Only one Button can be active at time.

Basic Usage

The active Button in the SelectionGrid is tracked through an integer. You must provide the integer as an argument in the function. To make the SelectionGrid interactive, you must assign the integer to the return value of the function. The number of elements in the content array that you provide will determine the number of Buttons that are shown in the SelectionGrid. You also can dictate the number of columns through the function arguments.

```
/* GUI.SelectionGrid example */

using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour
{
```

```
    private int selectionGridInt = 0;
    private string[] selectionStrings = {"Grid 1", "Grid 2", "Grid 3", "Grid 4"}

    void OnGUI ()
    {
        selectionGridInt = GUI.SelectionGrid (new Rect (25, 25, 300, 60), select

    }

}
```



The SelectionGrid created by the example code

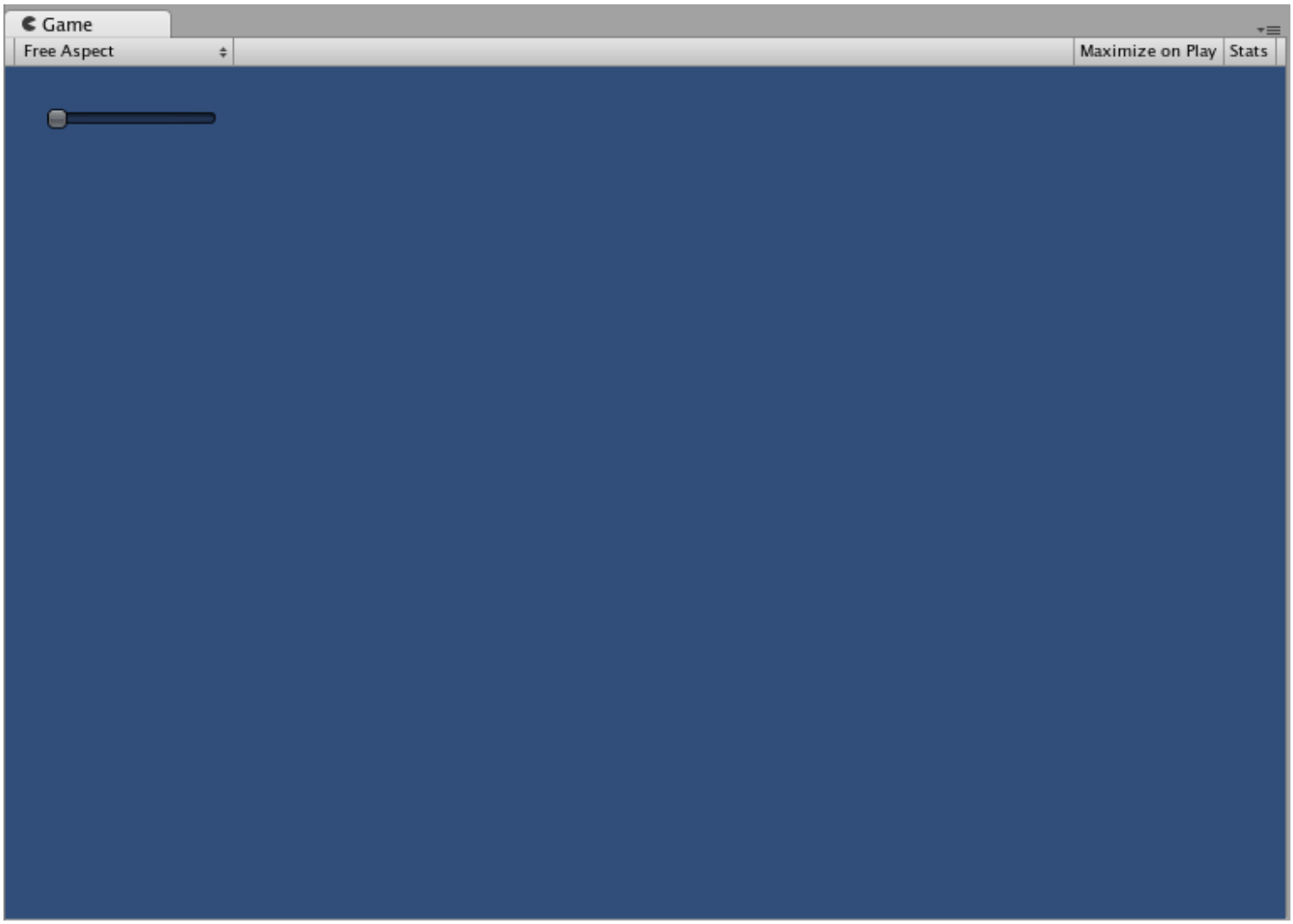
HorizontalSlider

The **HorizontalSlider** Control is a typical horizontal sliding knob that can be dragged to change a value between predetermined min and max values.

Basic Usage

The position of the Slider knob is stored as a float. To display the position of the knob, you provide that float as one of the arguments in the function. There are two additional values that determine the minimum and maximum values. If you want the slider knob to be adjustable, assign the slider value float to be the return value of the Slider function.

```
/* Horizontal Slider example */

using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour
{

    private float hSliderValue = 0.0f;

    void OnGUI ()
    {
        hSliderValue = GUI.HorizontalSlider (new Rect (25, 25, 100, 30), hSlider
    }

}
```



The Horizontal Slider created by the example code

VerticalSlider

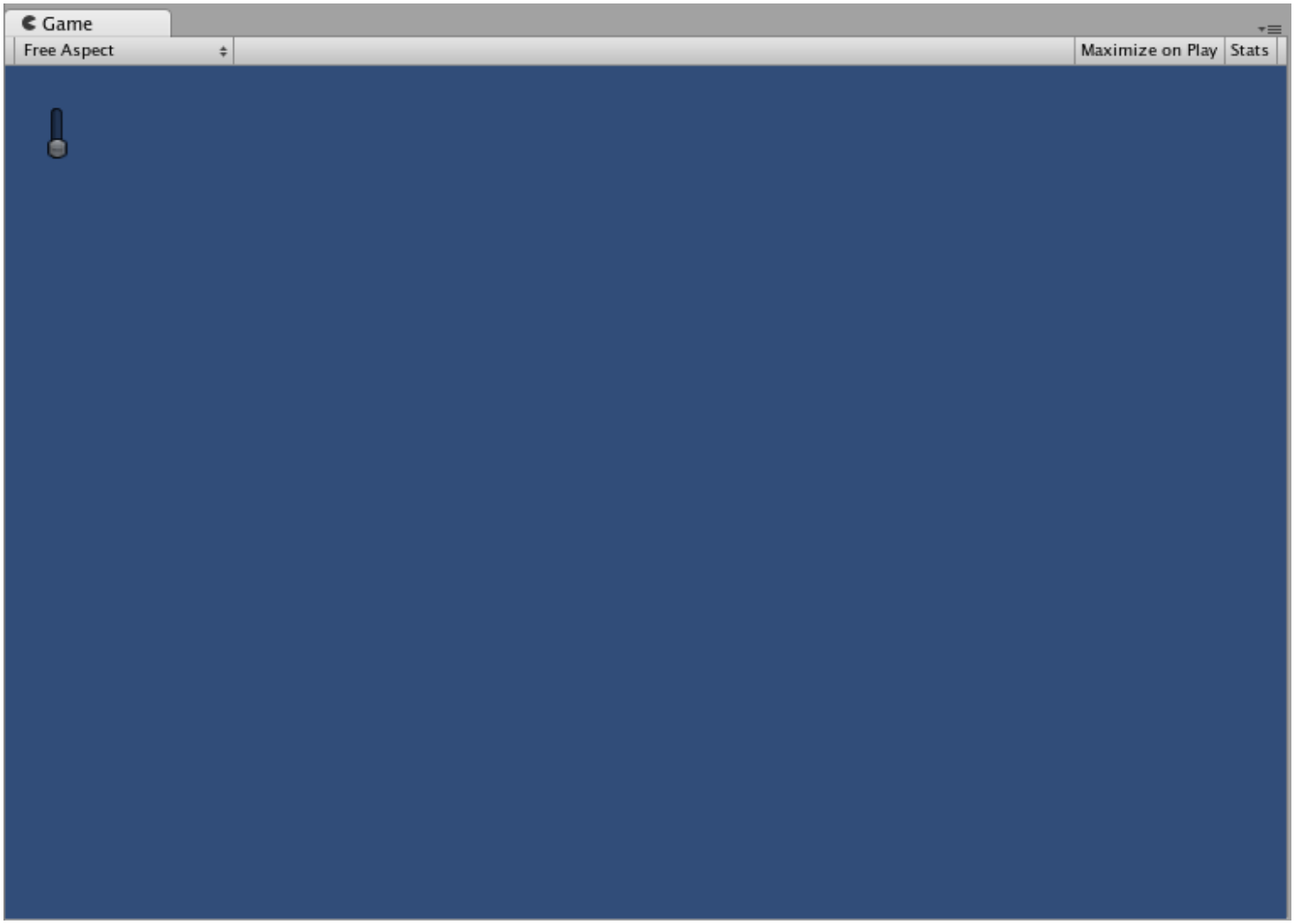
The **VerticalSlider** Control is a typical vertical sliding knob that can be dragged to change a value between predetermined min and max values.

Basic Usage

The position of the Slider knob is stored as a float. To display the position of the knob, you provide that float as one of the arguments in the function. There are two additional values that determine the minimum and maximum values. If you want the slider knob to be adjustable, assign the slider value float to be the return value of the Slider function.

```
/* Vertical Slider example */

using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour
{
```

```
    private float vSliderValue = 0.0f;

    void OnGUI ()
    {
        vSliderValue = GUI.VerticalSlider (new Rect (25, 25, 100, 30), vSliderVa
    }

}
```



The Vertical Slider created by the example code

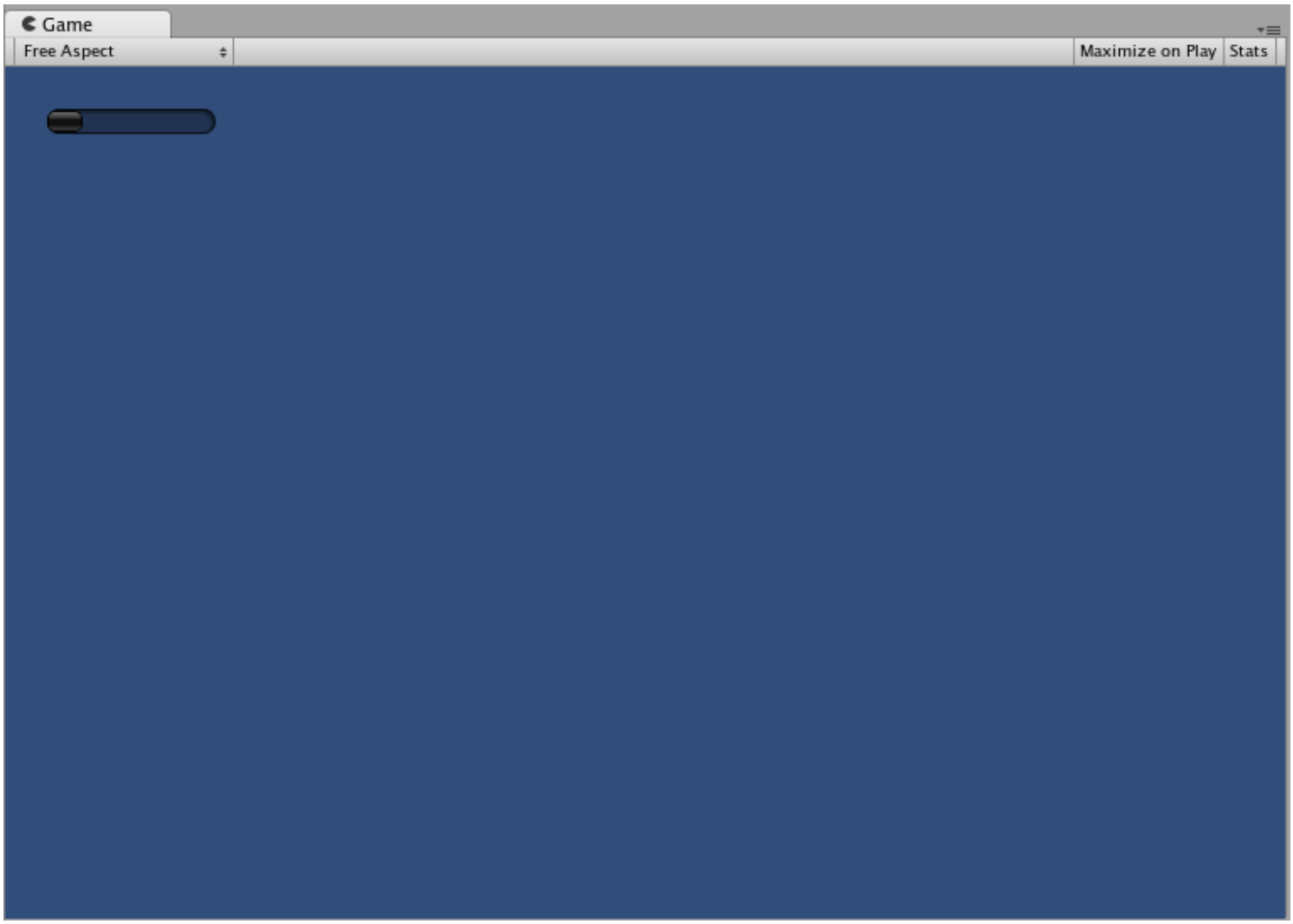
HorizontalScrollbar

The **HorizontalScrollbar** Control is similar to a **Slider** Control, but visually similar to Scrolling elements for web browsers or word processors. This control is used to navigate the **ScrollView** Control.

Basic Usage

Horizontal Scrollbars are implemented identically to Horizontal Sliders with one exception: There is an additional argument which controls the width of the Scrollbar knob itself.

```
/* Horizontal Scrollbar example */

using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour
{
    private float hScrollbarValue;

    void OnGUI ()
    {
        hScrollbarValue = GUI.HorizontalScrollbar (new Rect (25, 25, 100, 30), h
    }

}
```



The Horizontal Scrollbar created by the example code

VerticalScrollbar

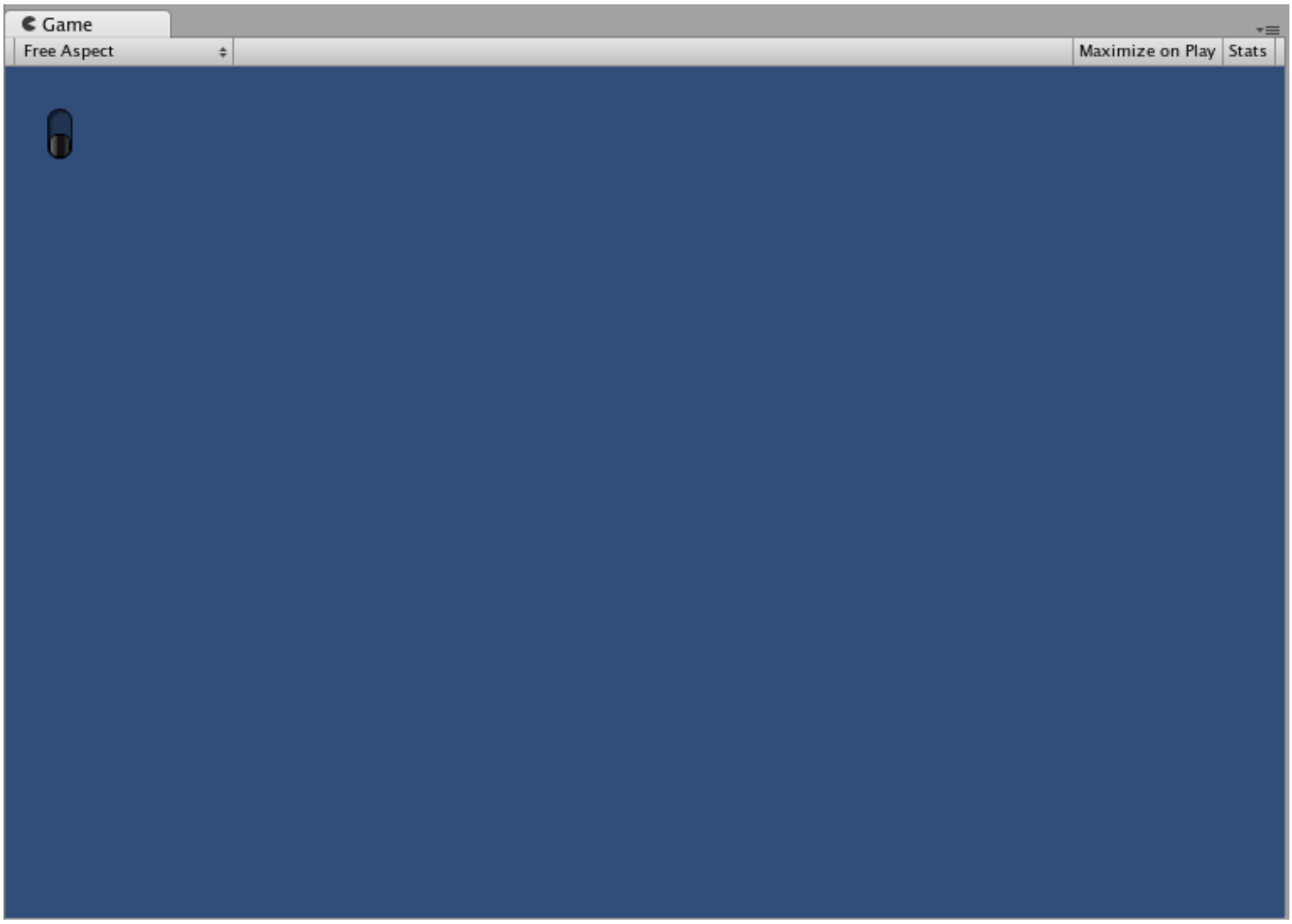
The **VerticalScrollbar** Control is similar to a **Slider** Control, but visually similar to Scrolling elements for web browsers or word processors. This control is used to navigate the **ScrollView** Control.

Basic Usage

Vertical Scrollbars are implemented identically to Vertical Sliders with one exception: There is an additional argument which controls the height of the Scrollbar knob itself.

```
/* Vertical Scrollbar example */

using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour
{

    private float vScrollbarValue;
```

```
    void OnGUI ()
    {
        vScrollbarValue = GUI. VerticalScrollbar (new Rect (25, 25, 100, 30), vS
    }
}
```



The Vertical Scrollbar created by the example code

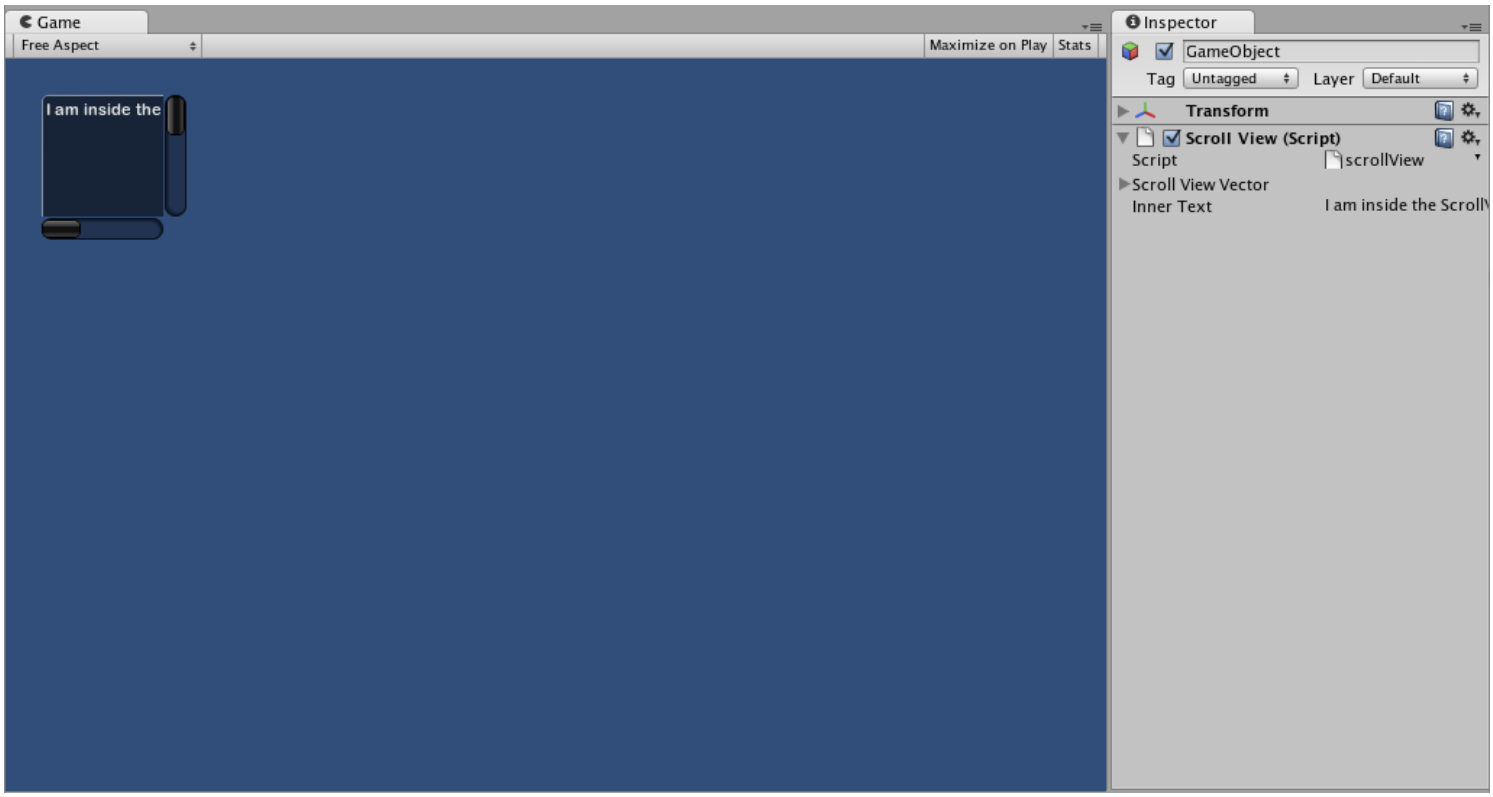
ScrollView

ScrollViews are Controls that display a viewable area of a much larger set of Controls.

Basic Usage

ScrollViews require two **Rects** as arguments. The first **Rect** defines the location and size of the viewable ScrollView area on the screen. The second **Rect** defines the size of the space contained inside the viewable area. If the space inside the viewable area is larger than the viewable area, Scrollbars will appear as appropriate. You must also assign and provide a 2D Vector which stores the position of the viewable area that is displayed.

```
/* ScrollView example */

using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour
{

    private Vector2 scrollViewVector = Vector2.zero;
    private string innerText = "I am inside the ScrollView";

    void OnGUI ()
    {
        // Begin the ScrollView
        scrollViewVector = GUI.BeginScrollView (new Rect (25, 25, 100, 100), scr

        // Put something inside the ScrollView
        innerText = GUI.TextArea (new Rect (0, 0, 400, 400), innerText);

        // End the ScrollView
        GUI.EndScrollView();
    }

}
```



The ScrollView created by the example code

Window

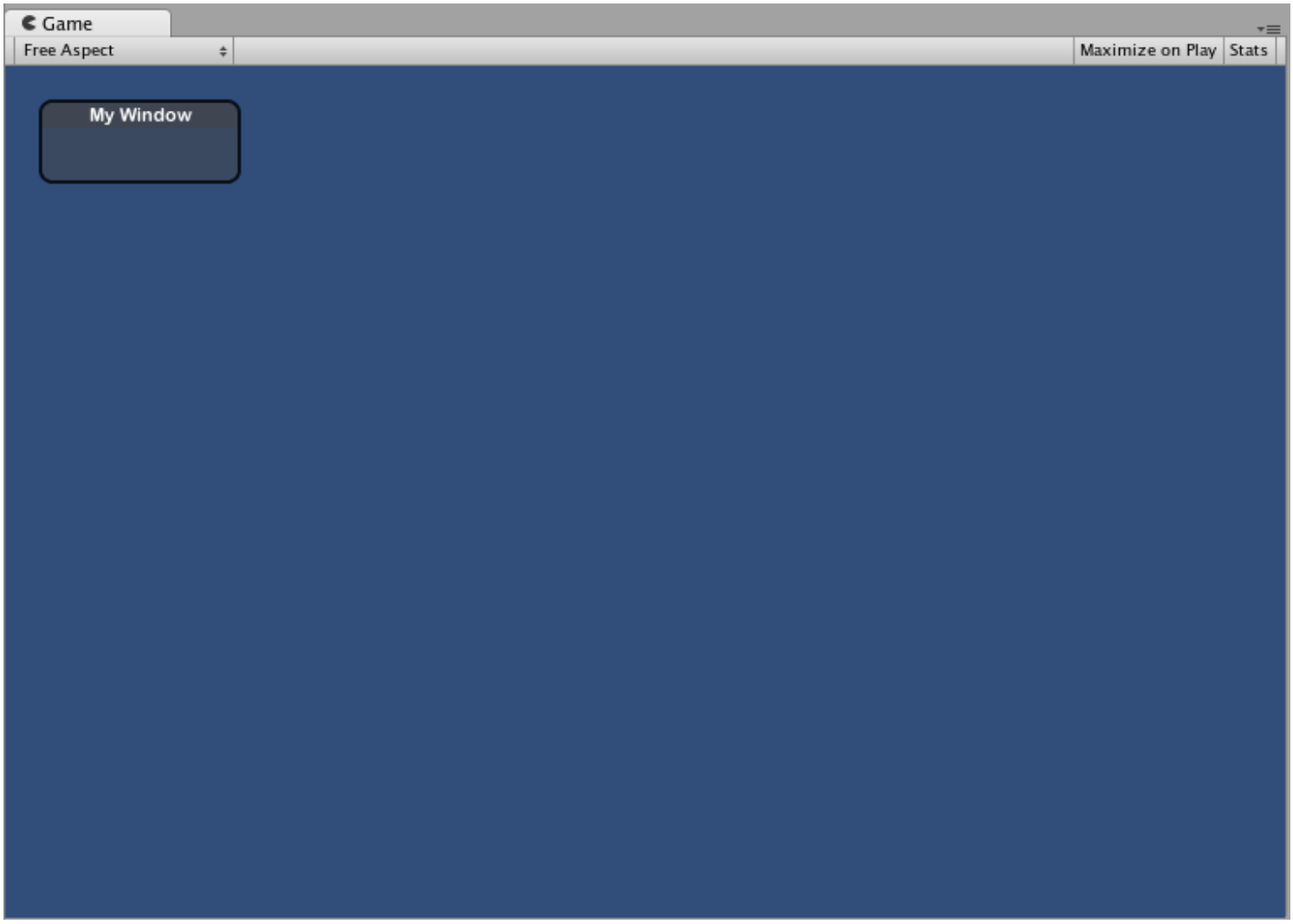
Windows are drag-able containers of Controls. They can receive and lose focus when clicked. Because of this, they are implemented slightly differently from the other Controls. Each Window has an **id** number, and its contents are declared inside a separate function that is called when the Window has focus.

Basic Usage

Windows are the only Control that require an additional function to work properly. You must provide an **id** number and a function name to be executed for the Window. Inside the Window function, you create your actual behaviors or contained Controls.

```
/* Window example */

using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour
{

    private Rect windowRect = new Rect (20, 20, 120, 50);

    void OnGUI ()
    {
        windowRect = GUI.Window (0, windowRect, WindowFunction, "My Window");
    }
```

```
    void WindowFunction (int windowID)
    {
        // Draw any Controls inside the window here
    }

}
```



The Window created by the example code

GUI.changed

To detect if the user did any action in the GUI (clicked a button, dragged a slider, etc), read the **GUI.changed** value from your script. This gets set to true when the user has done something, making it easy to validate the user input.

A common scenario would be for a Toolbar, where you want to change a specific value based on which Button in the Toolbar was clicked. You don't want to assign the value in every call to **OnGUI()**, only when one of the Buttons has been clicked.

```
/* GUI.changed example */

using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour
{

    private int selectedToolbar = 0;
    private string[] toolbarStrings = {"One", "Two"};

    void OnGUI ()
    {
        // Determine which button is active, whether it was clicked this frame o
        selectedToolbar = GUI.Toolbar (new Rect (50, 10, Screen.width - 100, 30)

        // If the user clicked a new Toolbar button this frame, we'll process th
        if (GUI.changed)
        {
            Debug.Log("The toolbar was clicked");

            if (0 == selectedToolbar)
            {
                Debug.Log("First button was clicked");
            }
            else
            {
                Debug.Log("Second button was clicked");
            }
        }
    }

}
```

GUI.changed will return true if any GUI Control placed before it was manipulated by the user.

Customization

Leave feedback

Customizing your IMGUI Controls

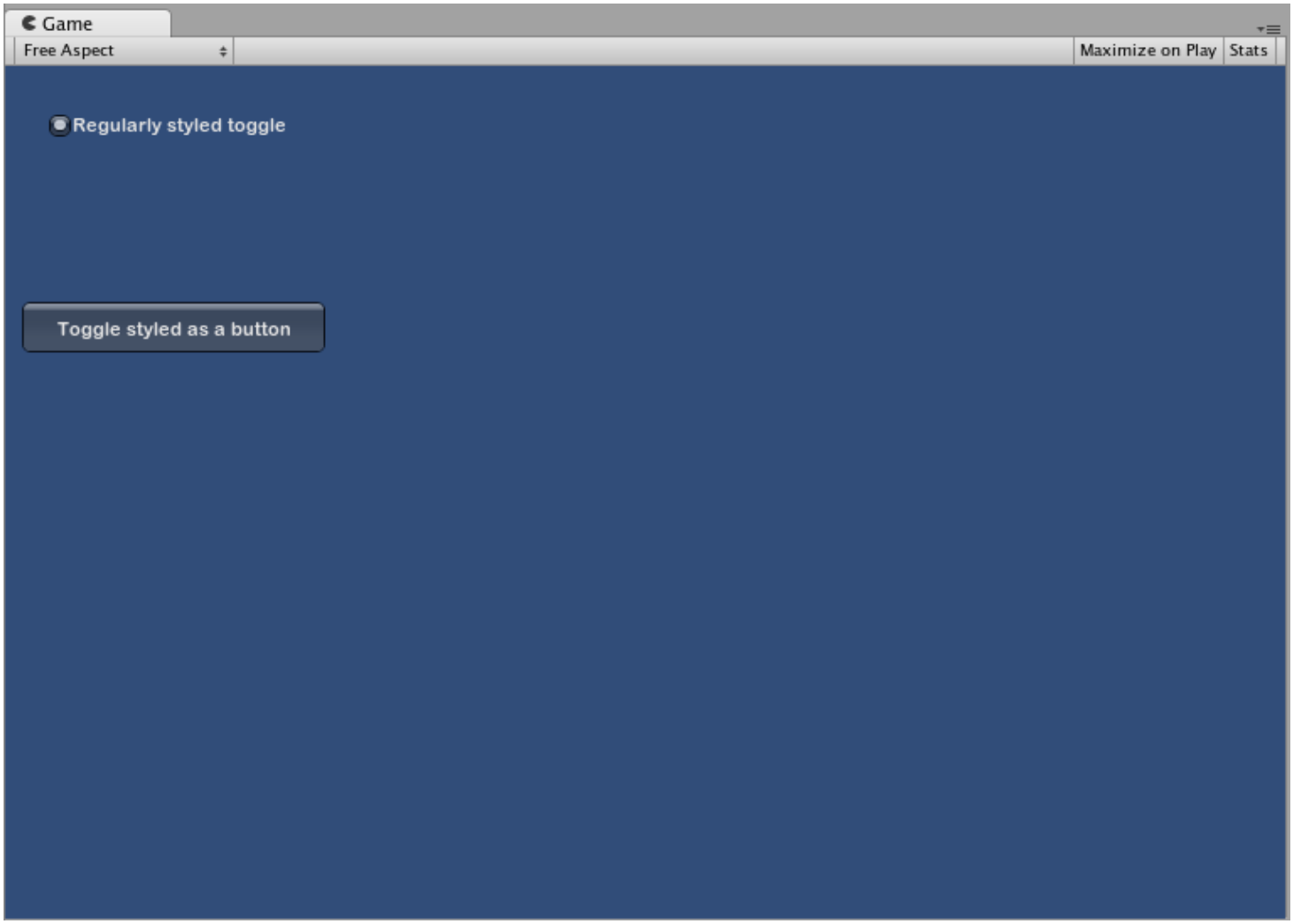
Although Unity's IMGUI system is mainly intended for creating developer tools and debugging interfaces, you can still customize and style them in many ways. In Unity's IMGUI system, you can fine-tune the appearance of your Controls with many details. Control appearances are dictated with **GUIStyles**. By default, when you create a Control without defining a GUIStyle, Unity's default GUIStyle is applied. This style is internal in Unity and can be used in published games for quick prototyping, or if you choose not to stylize your Controls.

When you have a large number of different GUIStyles to work with, you can define them all within a single **GUISkin**. A GUISkin is no more than a collection of GUIStyles.

How Styles change the look of your GUI Controls

GUIStyles are designed to mimic Cascading Style Sheets (CSS) for web browsers. Many different CSS methodologies have been adapted, including differentiation of individual state properties for styling, and separation between the content and the appearance.

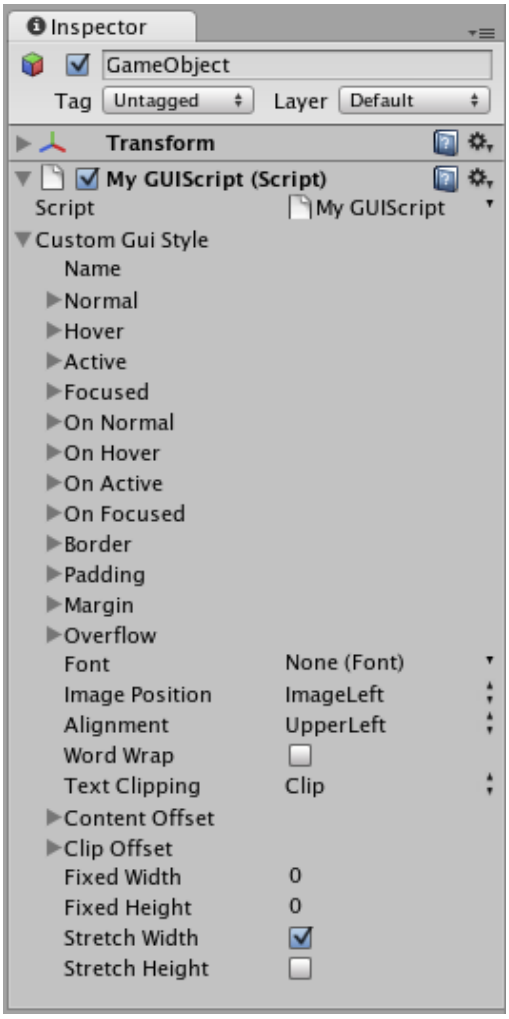
Where the Control defines the content, the Style defines the appearance. This allows you to create combinations like a functional **Toggle** which looks like a normal **Button**.



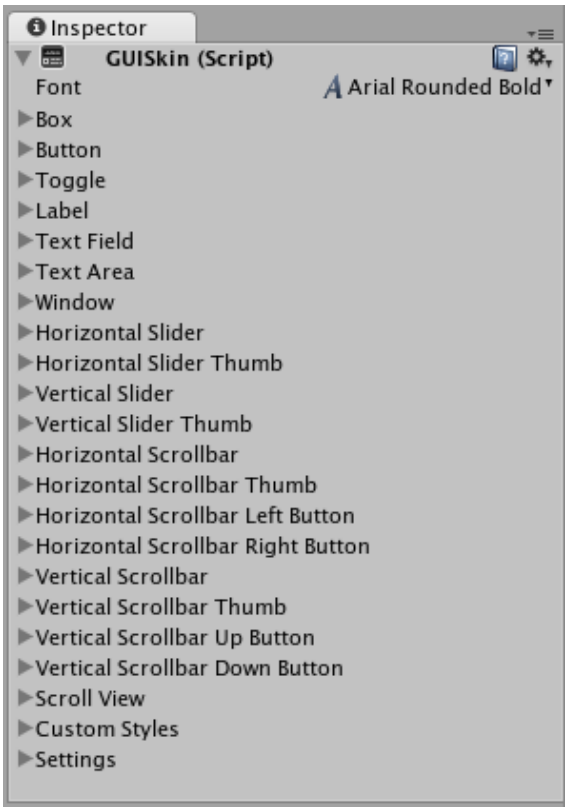
Two Toggle Controls styled differently

The difference between Skins and Styles

As stated earlier, GUISkins are a collection of GUIStyles. Styles define the appearance of a GUI Control. You do not have to use a Skin if you want to use a Style.



A single GUIStyle shown in the Inspector



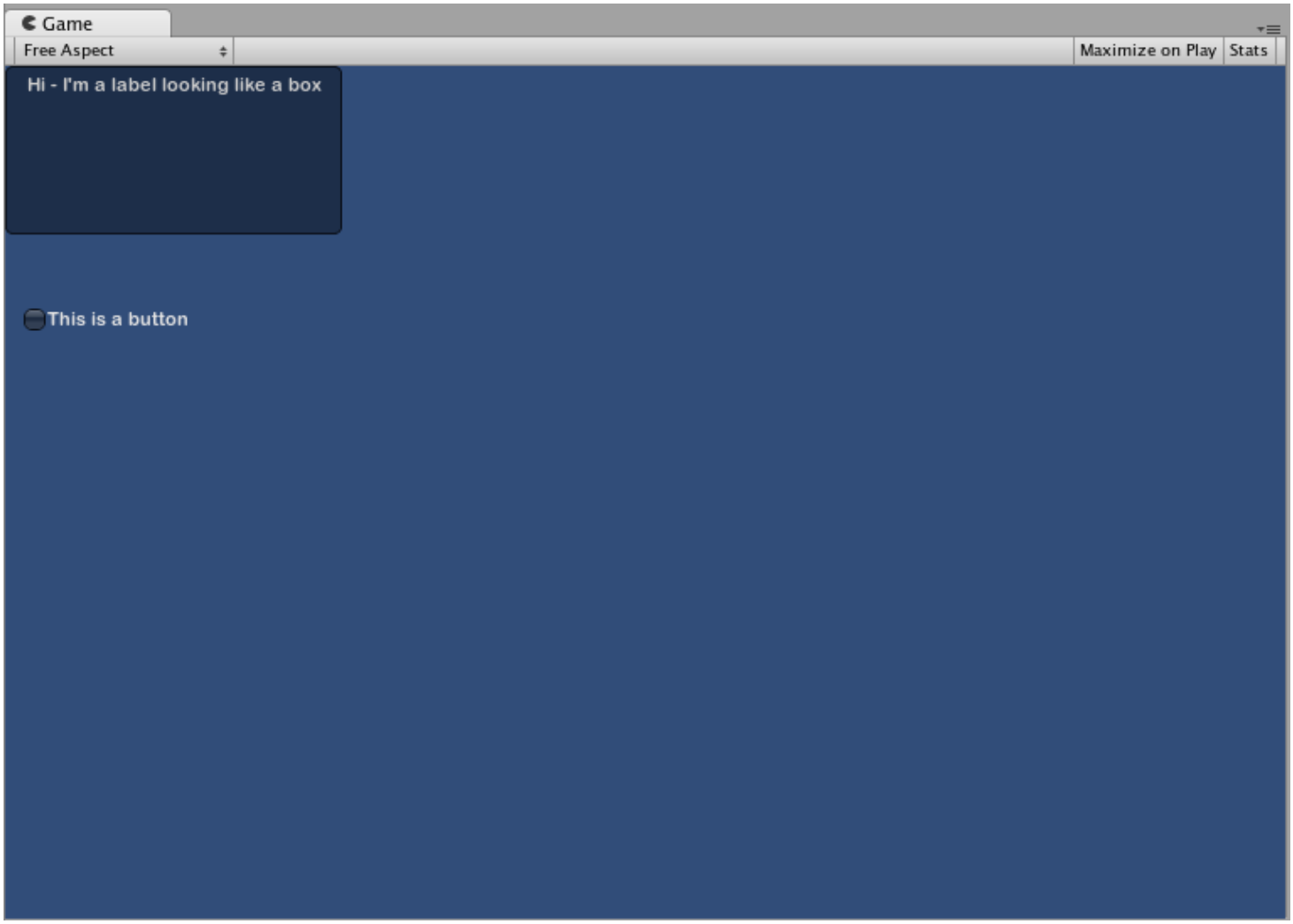
A single GUISkin shown in the Inspector - observe that it contains multiple GUIStyles

Working with Styles

All GUI Control functions have an optional last parameter: the GUIStyle to use for displaying the Control. If this is omitted, Unity's default GUIStyle will be used. This works internally by applying the name of the control type as a string, so **GUI.Button()** uses the "button" style, **GUI.Toggle()** uses the "toggle" style, **etc**. You can override the default GUIStyle for a control by specifying it as the last parameter.

```
/* Override the default Control Style with a different style in the UnityGUI default


// JavaScript
function OnGUI () {
    // Make a label that uses the "box" GUIStyle.
    GUI.Label (Rect (0,0,200,100), "Hi - I'm a label looking like a box", "box");

    // Make a button that uses the "toggle" GUIStyle
    GUI.Button (Rect (10,140,180,20), "This is a button", "toggle");
}


// C#
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour {
```

```
    void OnGUI () {
        // Make a label that uses the "box" GUIStyle.
        GUI.Label (new Rect (0,0,200,100), "Hi - I'm a label looking like a box", "bo

        // Make a button that uses the "toggle" GUIStyle
        GUI.Button (new Rect (10,140,180,20), "This is a button", "toggle");
    }

}
```



The controls created by the code example above

Making a public variable GUIStyle

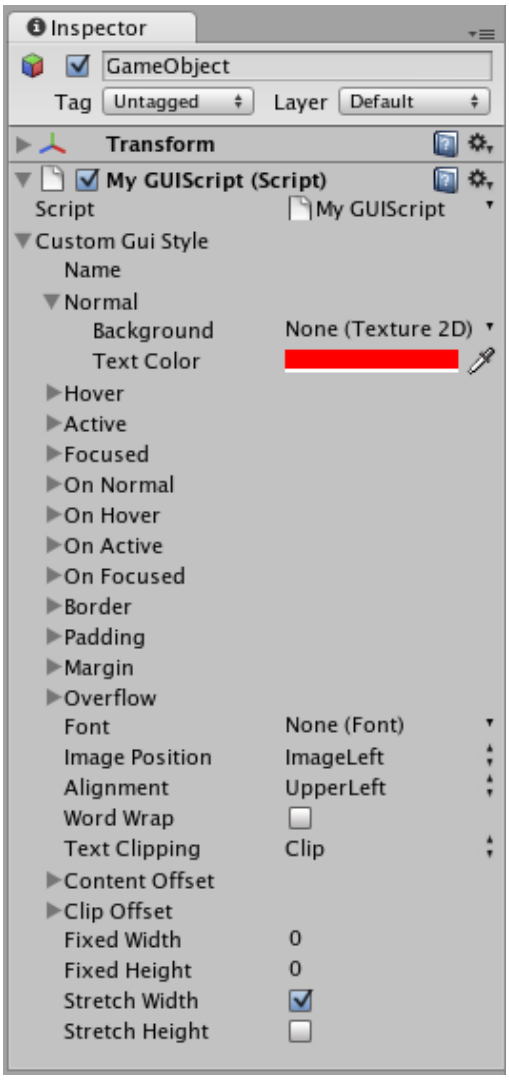
When you declare a public GUIStyle variable, all elements of the Style will show up in the **Inspector**. You can edit all of the different values there.

```
/* Overriding the default Control Style with one you've defined yourself */
```

```
// JavaScript
var customButton : GUIStyle;

function OnGUI () {
    // Make a button. We pass in the GUIStyle defined above as the style to use
    GUI.Button (Rect (10,10,150,20), "I am a Custom Button", customButton);
}


// C#
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour {

    public GUIStyle customButton;

    void OnGUI () {
        // Make a button. We pass in the GUIStyle defined above as the style to use
        GUI.Button (new Rect (10,10,150,20), "I am a Custom Button", customButton);
    }

}
```

Changing the different style elements

When you have declared a GUIStyle, you can modify that style in the Inspector. There are a great number of States you can define, and apply to any type of Control.



Styles are modified on a per-script, per-GameObject basis

Any Control State must be assigned a **Background** Color before the specified **Text Color** will be applied.

For more information about individual GUIStyles, please read the GUIStyle Component Reference page.

Working with Skins

For more complicated GUI systems, it makes sense to keep a collection of styles in one place. This is what a GUISkin does. A GUISkin contains multiple different Styles, essentially providing a complete face-lift to all GUI Controls.

Creating a new GUISkin

To create a GUISkin, select **Assets->Create->GUI Skin** from the menu bar. This will create a GUI Skin in your Project Folder. Select it to see all GUIStyles defined by the Skin in the Inspector.

Applying the skin to a GUI

To use a skin you've created, assign it to **GUI.skin** in your **OnGUI()** function.

```
/* Make a property containing a reference to the skin you want to use */


// JavaScript
```

```
var mySkin : GUISkin;

function OnGUI () {
    // Assign the skin to be the one currently used.
    GUI.skin = mySkin;

    // Make a button. This will get the default "button" style from the skin assigned
    GUI.Button (Rect (10,10,150,20), "Skinned Button");
}


// C#
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour {

    public GUISkin mySkin;

    void OnGUI () {
        // Assign the skin to be the one currently used.
        GUI.skin = mySkin;

        // Make a button. This will get the default "button" style from the skin assi
        GUI.Button (new Rect (10,10,150,20), "Skinned Button");
    }

}
```

You can switch skins as much as you like throughout a single **OnGUI()** call.

```
/* Example of switching skins in the same OnGUI() call */


// JavaScript
var mySkin : GUISkin;

var toggle = true;

function OnGUI () {
    // Assign the skin to be the one currently used.
    GUI.skin = mySkin;

    // Make a toggle. This will get the "button" style from the skin assigned to mySk
```

```
    toggle = GUI.Toggle (Rect (10,10,150,20), toggle, "Skinned Button", "button");

    // Assign the currently skin to be Unity's default.
    GUI.skin = null;

    // Make a button. This will get the default "button" style from the built-in skin
    GUI.Button (Rect (10,35,150,20), "Built-in Button");
}


// C#
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour {

    public GUISkin mySkin;
    private bool toggle = true;

    void OnGUI () {
        // Assign the skin to be the one currently used.
        GUI.skin = mySkin;

        // Make a toggle. This will get the "button" style from the skin assigned to
        toggle = GUI.Toggle (new Rect (10,10,150,20), toggle, "Skinned Button", "butt

        // Assign the currently skin to be Unity's default.
        GUI.skin = null;

        // Make a button. This will get the default "button" style from the built-in
        GUI.Button (new Rect (10,35,150,20), "Built-in Button");
    }

}
```

Changing GUI Font Size

This example will show you how to dynamically change the font size through code.

First create a new project in Unity. Then make a C# script called Fontsize.cs and paste the following code in:

```
// C# example
using UnityEngine;
```

```
using System.Collections;

public class Fontsize : MonoBehaviour
{
    void OnGUI ()
    {
        //Set the GUIStyle style to be label
        GUIStyle style = GUI.skin.GetStyle ("label");

        //Set the style font size to increase and decrease over time
        style.fontSize = (int)(20.0f + 10.0f * Mathf.Sin (Time.time));

        //Create a label and display with the current settings
        GUI.Label (new Rect (10, 10, 200, 80), "Hello World!");
    }

}
```

Save the script and attach it to an empty **GameObject**, click play to see the font loop through increasing and decreasing in size over time. You may notice that the font does not smoothly change size, this is becauses there is not an infinite number of font sizes.

This specific example requires that the default font (Arial) is loaded and marked as dynamic. You cannot change the size of any font that is not marked as dynamic.

IMGUI Layout Modes

Leave feedback

Fixed Layout vs Automatic Layout

There are two different modes you can use to arrange and organize your **UI** when using the IMGUI system: Fixed and Automatic. Up until now, every IMGUI example provided in this guide has used Fixed Layout. To use Automatic Layout, write **GUILayout** instead of **GUI** when calling control functions. You do not have to use one Layout mode over the other, and you can use both modes at once in the same **OnGUI()** function.

Fixed Layout makes sense to use when you have a pre-designed interface to work from. Automatic Layout makes sense to use when you don't know how many elements you need up front, or don't want to worry about hand-positioning each Control. For example, if you are creating a number of different buttons based on Save Game files, you don't know exactly how many buttons will be drawn. In this case Automatic Layout might make more sense. It is really dependent on the design of your game and how you want to present your interface.

There are two key differences when using Automatic Layout:

GUILayout is used instead of **GUI**
No **Rect()** function is required for Automatic Layout Controls

```
/* Two key differences when using Automatic Layout */


// JavaScript
function OnGUI () {
    // Fixed Layout
    GUI.Button (Rect (25,25,100,30), "I am a Fixed Layout Button");

    // Automatic Layout
    GUILayout.Button ("I am an Automatic Layout Button");
}


// C#
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour {

    void OnGUI () {
        // Fixed Layout
        GUI.Button (new Rect (25,25,100,30), "I am a Fixed Layout Button");

        // Automatic Layout
        GUILayout.Button ("I am an Automatic Layout Button");
    }
```

```
}
```

Arranging Controls

Depending on which Layout Mode you're using, there are different hooks for controlling where your Controls are positioned and how they are grouped together. In Fixed Layout, you can put different Controls into **Groups**. In Automatic Layout, you can put different Controls into **Areas**, **Horizontal Groups**, and **Vertical Groups**

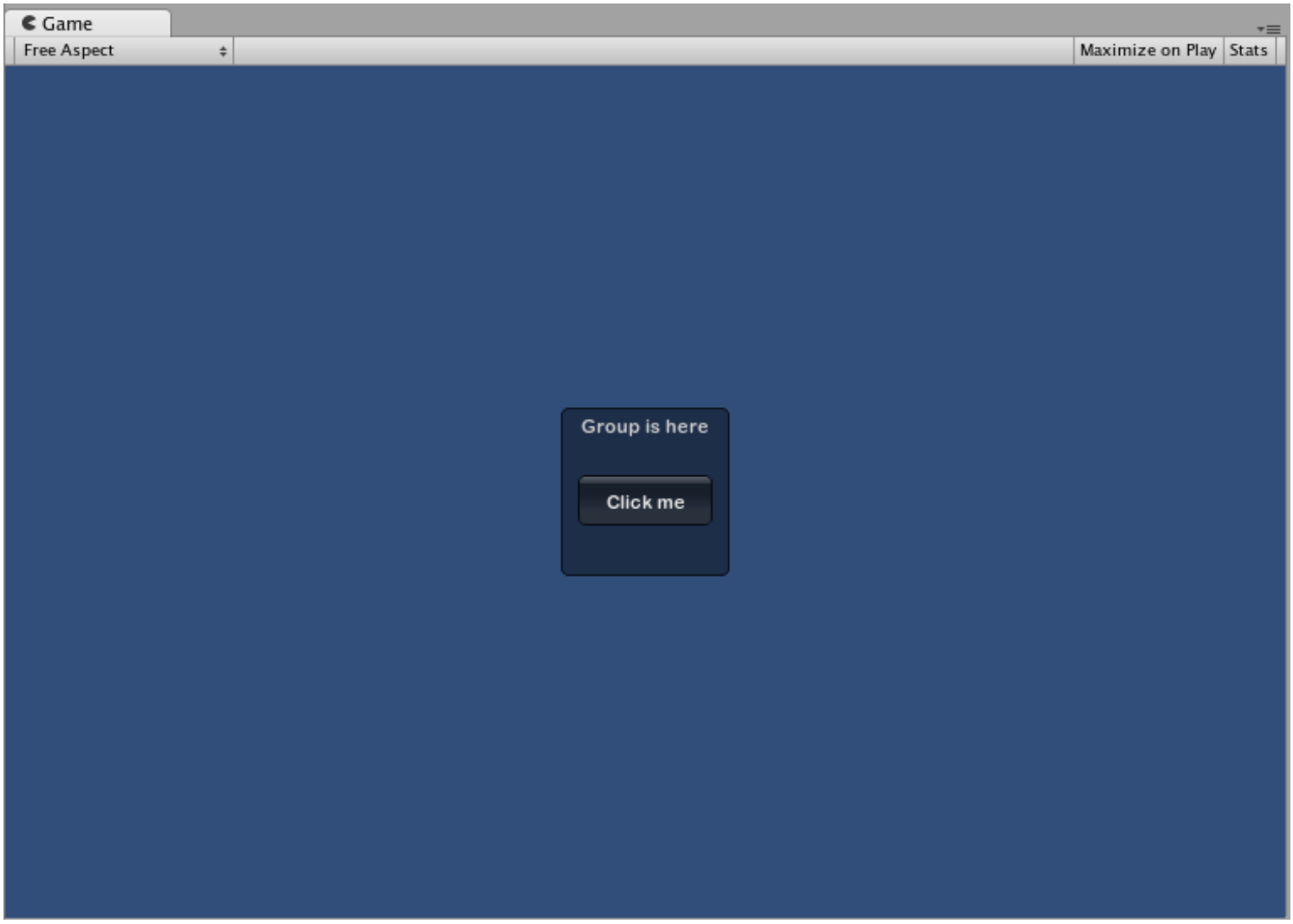
Fixed Layout - Groups

Groups are a convention available in Fixed Layout Mode. They allow you to define areas of the screen that contain multiple Controls. You define which Controls are inside a Group by using the **GUI.BeginGroup()** and **GUI.EndGroup()** functions. All Controls inside a Group will be positioned based on the Group's top-left corner instead of the screen's top-left corner. This way, if you reposition the group at runtime, the relative positions of all Controls in the group will be maintained.

As an example, it's very easy to center multiple Controls on-screen.

```
/* Center multiple Controls on the screen using Groups */


// JavaScript
function OnGUI () {
    // Make a group on the center of the screen
    GUI.BeginGroup (Rect (Screen.width / 2 - 50, Screen.height / 2 - 50, 100, 10
    // All rectangles are now adjusted to the group. (0,0) is the topleft corner

    // We'll make a box so you can see where the group is on-screen.
    GUI.Box (Rect (0,0,100,100), "Group is here");
    GUI.Button (Rect (10,40,80,30), "Click me");

    // End the group we started above. This is very important to remember!
    GUI.EndGroup ();
}


// C#
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour {
```

```
    void OnGUI () {
        // Make a group on the center of the screen
        GUI.BeginGroup (new Rect (Screen.width / 2 - 50, Screen.height / 2 - 50,
        // All rectangles are now adjusted to the group. (0,0) is the topleft co

        // We'll make a box so you can see where the group is on-screen.
        GUI.Box (new Rect (0,0,100,100), "Group is here");
        GUI.Button (new Rect (10,40,80,30), "Click me");

        // End the group we started above. This is very important to remember!
        GUI.EndGroup ();
    }

}
```

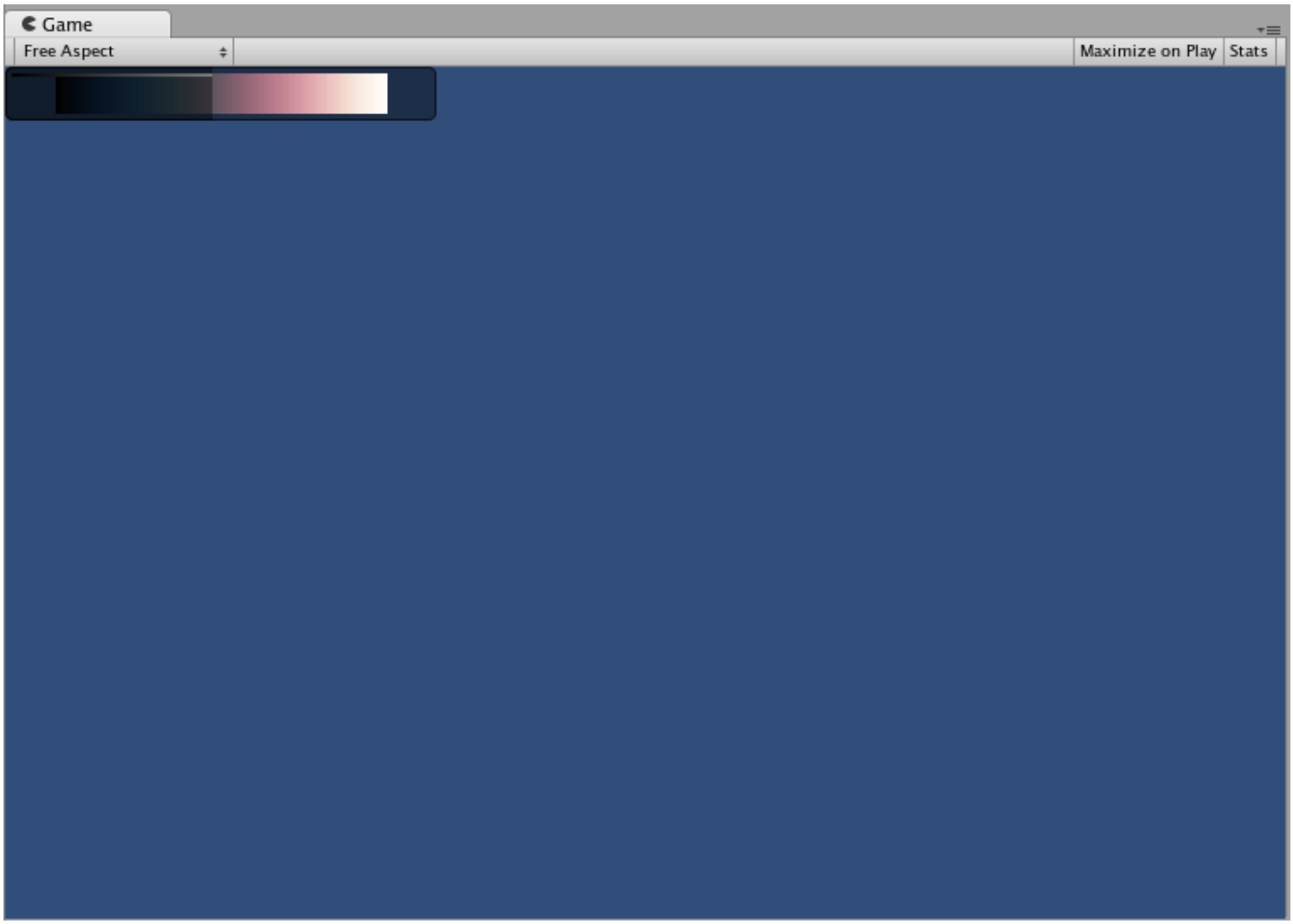


The above example centers controls regardless of the screen resolution

You can also nest multiple Groups inside each other. When you do this, each group has its contents clipped to its parent's space.

```
/* Using multiple Groups to clip the displayed Contents */


// JavaScript
var bgImage : Texture2D; // background image that is 256 x 32
var fgImage : Texture2D; // foreground image that is 256 x 32
var playerEnergy = 1.0; // a float between 0.0 and 1.0

function OnGUI () {
    // Create one Group to contain both images
    // Adjust the first 2 coordinates to place it somewhere else on-screen
    GUI.BeginGroup (Rect (0,0,256,32));

    // Draw the background image
    GUI.Box (Rect (0,0,256,32), bgImage);

    // Create a second Group which will be clipped
    // We want to clip the image and not scale it, which is why we need the seco
    GUI.BeginGroup (Rect (0,0,playerEnergy * 256, 32));

    // Draw the foreground image
    GUI.Box (Rect (0,0,256,32), fgImage);

    // End both Groups
    GUI.EndGroup ();
    GUI.EndGroup ();
}


// C#
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour {

    // background image that is 256 x 32
    public Texture2D bgImage;

    // foreground image that is 256 x 32
    public Texture2D fgImage;

    // a float between 0.0 and 1.0
```

```
    public float playerEnergy = 1.0f;

    void OnGUI () {
        // Create one Group to contain both images
        // Adjust the first 2 coordinates to place it somewhere else on-screen
        GUI.BeginGroup (new Rect (0,0,256,32));

        // Draw the background image
        GUI.Box (new Rect (0,0,256,32), bgImage);

            // Create a second Group which will be clipped
            // We want to clip the image and not scale it, which is why we need
            GUI.BeginGroup (new Rect (0,0,playerEnergy * 256, 32));

            // Draw the foreground image
            GUI.Box (new Rect (0,0,256,32), fgImage);

            // End both Groups
            GUI.EndGroup ();

        GUI.EndGroup ();
    }

}
```



You can nest Groups together to create clipping behaviors

Automatic Layout - Areas

Areas are used in Automatic Layout mode only. They are similar to Fixed Layout Groups in functionality, as they define a finite portion of the screen to contain GUILayout Controls. Because of the nature of Automatic Layout, you will nearly always use Areas.

In Automatic Layout mode, you do not define the area of the screen where the Control will be drawn at the Control level. The Control will automatically be placed at the upper-leftmost point of its containing area. This might be the screen. You can also create manually-positioned Areas. GUILayout Controls inside an area will be placed at the upper-leftmost point of that area.

```
/* A button placed in no area, and a button placed in an area halfway across the


// JavaScript
function OnGUI () {
    GUILayout.Button ("I am not inside an Area");
    GUILayout.BeginArea (Rect (Screen.width/2, Screen.height/2, 300, 300));
    GUILayout.Button ("I am completely inside an Area");
    GUILayout.EndArea ();
```

```
}


// C#
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour {

    void OnGUI () {
        GUILayout.Button ("I am not inside an Area");
        GUILayout.BeginArea (new Rect (Screen.width/2, Screen.height/2, 300, 300
        GUILayout.Button ("I am completely inside an Area");
        GUILayout.EndArea ();
    }

}
```

Notice that inside an Area, Controls with visible elements like Buttons and Boxes will stretch their width to the full length of the Area.

Automatic Layout - Horizontal and Vertical Groups

When using Automatic Layout, Controls will by default appear one after another from top to bottom. There are plenty of occasions you will want finer level of control over where your Controls are placed and how they are arranged. If you are using the Automatic Layout mode, you have the option of Horizontal and Vertical Groups.

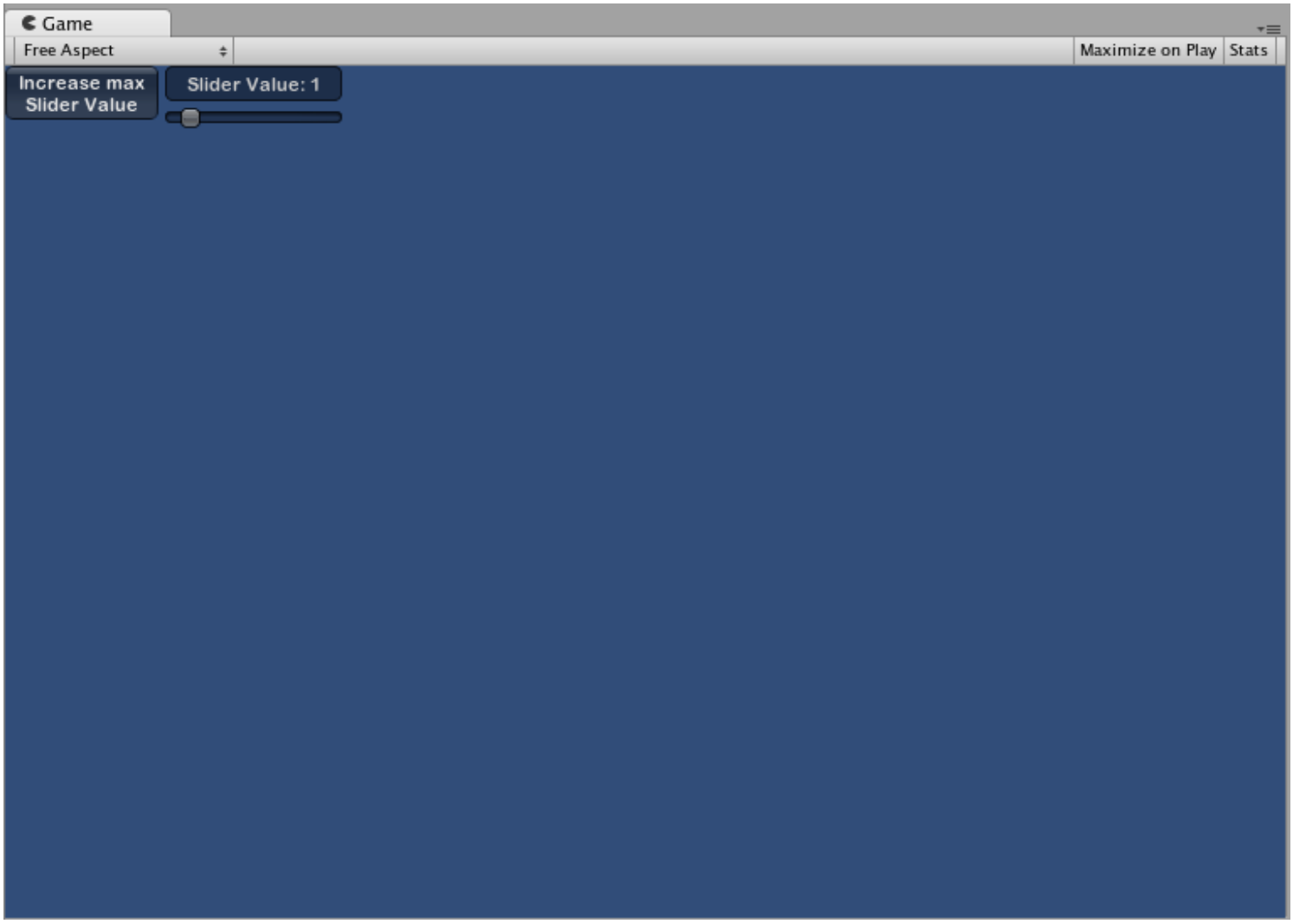
Like the other layout Controls, you call separate functions to start or end these groups. The specific functions are **GUILayout.BeginHoriztontal()**, **GUILayout.EndHorizontal()**, **GUILayout.BeginVertical()**, and **GUILayout.EndVertical()**.

Any Controls inside a Horizontal Group will always be laid out horizontally. Any Controls inside a Vertical Group will always be laid out vertically. This sounds plain until you start nesting groups inside each other. This allows you to arrange any number of controls in any imaginable configuration.

```
/* Using nested Horizontal and Vertical Groups */


// JavaScript
var sliderValue = 1.0;
var maxSliderValue = 10.0;
```

```
function OnGUI()
{
    // Wrap everything in the designated GUI Area
    GUILayout.BeginArea (Rect (0,0,200,60));

    // Begin the singular Horizontal Group
    GUILayout.BeginHorizontal();

    // Place a Button normally
    if (GUILayout.RepeatButton ("Increase max\nSlider Value"))
    {
        maxSliderValue += 3.0 * Time.deltaTime;
    }

    // Arrange two more Controls vertically beside the Button
    GUILayout.BeginVertical();
    GUILayout.Box("Slider Value: " + Mathf.Round(sliderValue));
    sliderValue = GUILayout.HorizontalSlider (sliderValue, 0.0, maxSliderValue);

    // End the Groups and Area
    GUILayout.EndVertical();
    GUILayout.EndHorizontal();
    GUILayout.EndArea();
}


// C#
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour {

    private float sliderValue = 1.0f;
    private float maxSliderValue = 10.0f;

    void OnGUI()
    {
        // Wrap everything in the designated GUI Area
        GUILayout.BeginArea (new Rect (0,0,200,60));

        // Begin the singular Horizontal Group
        GUILayout.BeginHorizontal();

        // Place a Button normally
        if (GUILayout.RepeatButton ("Increase max\nSlider Value"))
        {
```

```
            maxSliderValue += 3.0f * Time.deltaTime;
        }

        // Arrange two more Controls vertically beside the Button
        GUILayout.BeginVertical();
        GUILayout.Box("Slider Value: " + Mathf.Round(sliderValue));
        sliderValue = GUILayout.HorizontalSlider (sliderValue, 0.0f, maxSliderVa

        // End the Groups and Area
        GUILayout.EndVertical();
        GUILayout.EndHorizontal();
        GUILayout.EndArea();
    }

}
```



Three Controls arranged with Horizontal & Vertical Groups

Using GUILayoutOptions to define some controls

You can use GUILayoutOptions to override some of the Automatic Layout parameters. You do this by providing the options as the final parameters of the GUILayout Control.

Remember in the Areas example above, where the button stretches its width to 100% of the Area width? We can override that if we want to.

```
/* Using GUILayoutOptions to override Automatic Layout Control properties */


//JavaScript
function OnGUI () {
    GUILayout.BeginArea (Rect (100, 50, Screen.width-200, Screen.height-100));
    GUILayout.Button ("I am a regular Automatic Layout Button");
    GUILayout.Button ("My width has been overridden", GUILayout.Width (95));
    GUILayout.EndArea ();
}


// C#
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour {

    void OnGUI () {
        GUILayout.BeginArea (new Rect (100, 50, Screen.width-200, Screen.height-
        GUILayout.Button ("I am a regular Automatic Layout Button");
        GUILayout.Button ("My width has been overridden", GUILayout.Width (95));
        GUILayout.EndArea ();
    }

}
```

For a full list of possible GUILayoutOptions, please read the GUILayoutOption Scripting Reference page.

Extending IMGUI

Leave feedback

There are a number of ways to leverage and extend the IMGUI system to meet your needs. Controls can be mixed and created, and you have a lot of leverage in dictating how user input into the GUI is processed.

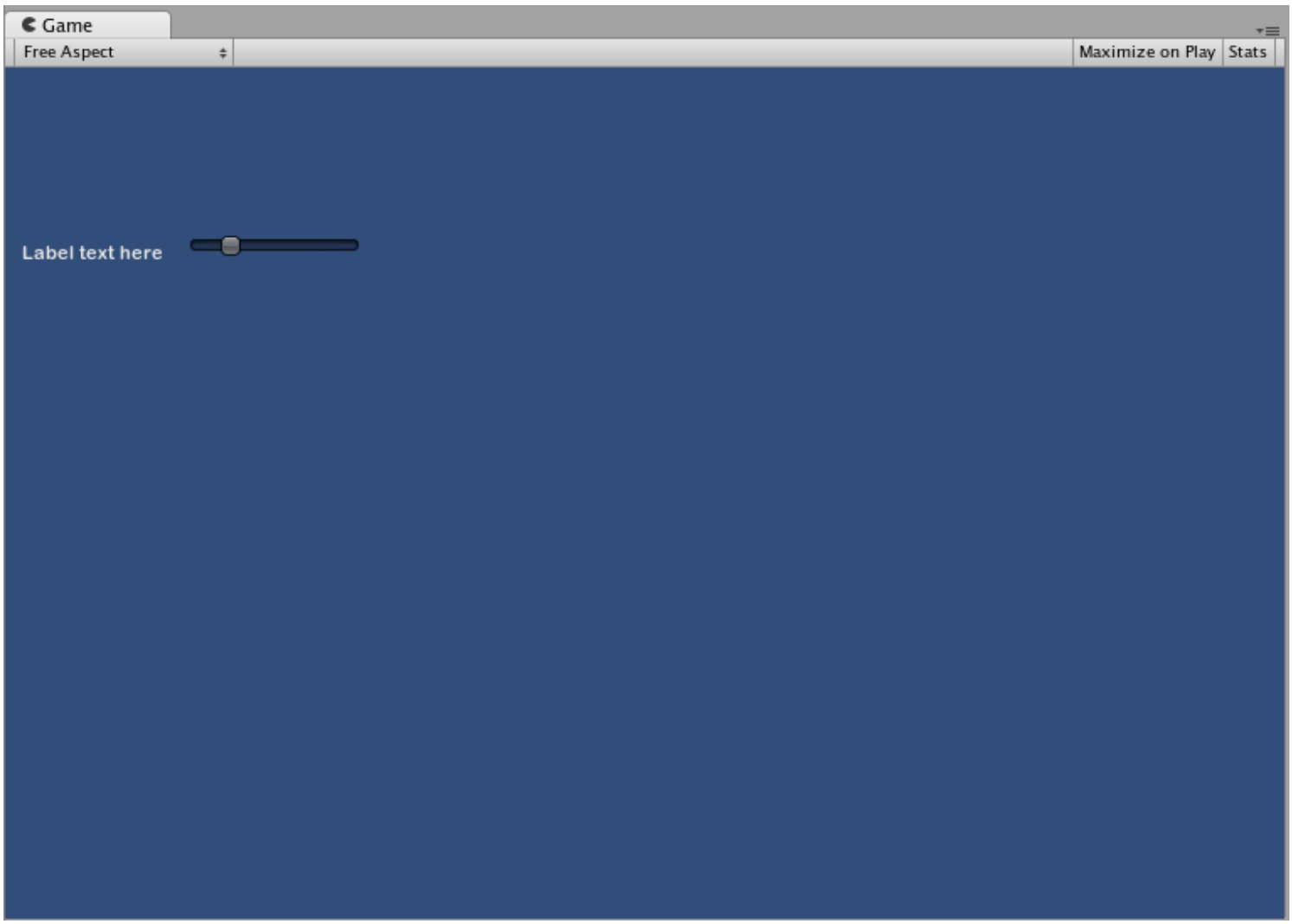
Compound Controls

There might be situations in your GUI where two types of Controls always appear together. For example, maybe you are creating a Character Creation screen, with several Horizontal Sliders. All of those Sliders need a Label to identify them, so the player knows what they are adjusting. In this case, you could partner every call to **GUI.Label()** with a call to **GUI.HorizontalSlider()**, or you could create a **Compound Control** which includes both a Label and a Slider together.

```
/* Label and Slider Compound Control */


// JavaScript
var mySlider : float = 1.0;

function OnGUI () {
    mySlider = LabelSlider (Rect (10, 100, 100, 20), mySlider, 5.0, "Label text
}

function LabelSlider (screenRect : Rect, sliderValue : float, sliderMaxValue : f
    GUI.Label (screenRect, labelText);
    screenRect.x += screenRect.width; // <- Push the Slider to the end of the La
    sliderValue = GUI.HorizontalSlider (screenRect, sliderValue, 0.0, sliderMaxV
    return sliderValue;
}


// C#
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour {

    private float mySlider = 1.0f;

    void OnGUI () {
        mySlider = LabelSlider (new Rect (10, 100, 100, 20), mySlider, 5.0f, "La
    }

    float LabelSlider (Rect screenRect, float sliderValue, float sliderMaxValue,
        GUI.Label (screenRect, labelText);
```

```
        // <- Push the Slider to the end of the Label
        screenRect.x += screenRect.width;

        sliderValue = GUI.HorizontalSlider (screenRect, sliderValue, 0.0f, slide
        return sliderValue;
    }

}
```

In this example, calling **LabelSlider()** and passing the correct arguments will provide a Label paired with a Horizontal Slider. When writing Compound Controls, you have to remember to return the correct value at the end of the function to make it interactive.



The above Compound Control always creates this pair of Controls

Static Compound Controls

By using **Static** functions, you can create an entire collection of your own Compound Controls that are self-contained. This way, you do not have to declare your function in the same script you want to use it.

```
/* This script is called CompoundControls */


// JavaScript
static function LabelSlider (screenRect : Rect, sliderValue : float, sliderMaxVa
    GUI.Label (screenRect, labelText);
    screenRect.x += screenRect.width; // <- Push the Slider to the end of the La
    sliderValue = GUI.HorizontalSlider (screenRect, sliderValue, 0.0, sliderMaxV
    return sliderValue;
}


// C#
using UnityEngine;
using System.Collections;

public class CompoundControls : MonoBehaviour {

    public static float LabelSlider (Rect screenRect, float sliderValue, float s
        GUI.Label (screenRect, labelText);

        // <- Push the Slider to the end of the Label
        screenRect.x += screenRect.width;

        sliderValue = GUI.HorizontalSlider (screenRect, sliderValue, 0.0f, slide
        return sliderValue;
    }

}
```

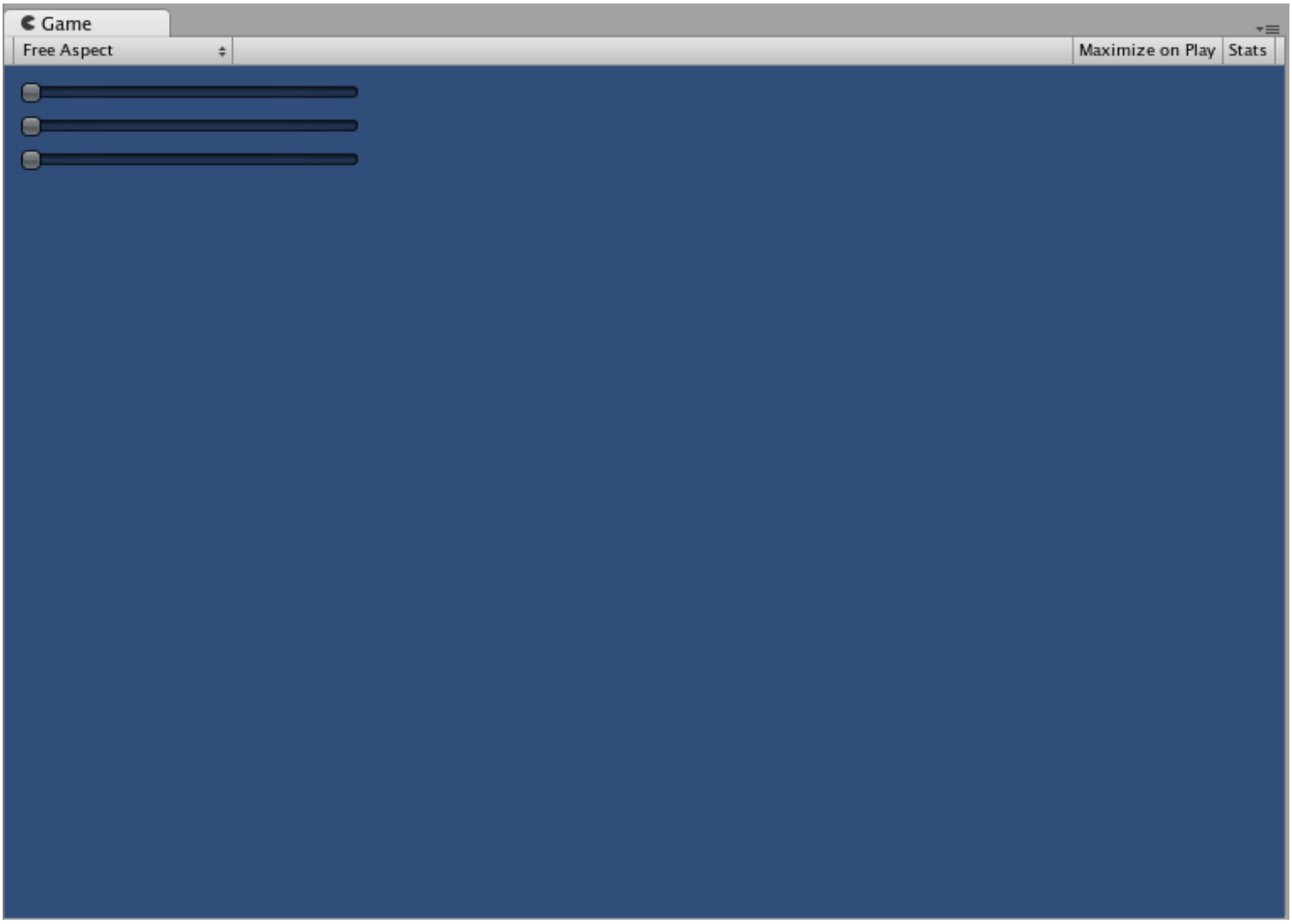
By saving the above example in a script called **CompoundControls**, you can call the **LabelSlider()** function from any other script by simply typing **CompoundControls.LabelSlider()** and providing your arguments.

Elaborate Compound Controls

You can get very creative with Compound Controls. They can be arranged and grouped in any way you like. The following example creates a re-usable RGB Slider.

```
/* RGB Slider Compound Control */


// JavaScript
var myColor : Color;

function OnGUI () {
    myColor = RGBSlider (Rect (10,10,200,10), myColor);
}

function RGBSlider (screenRect : Rect, rgb : Color) : Color {
    rgb.r = GUI.HorizontalSlider (screenRect, rgb.r, 0.0, 1.0);
    screenRect.y += 20; // <- Move the next control down a bit to avoid overlapp
    rgb.g = GUI.HorizontalSlider (screenRect, rgb.g, 0.0, 1.0);
    screenRect.y += 20; // <- Move the next control down a bit to avoid overlapp
    rgb.b = GUI.HorizontalSlider (screenRect, rgb.b, 0.0, 1.0);
    return rgb;
}


// C#
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour {

    public Color myColor;

    void OnGUI () {
        myColor = RGBSlider (new Rect (10,10,200,10), myColor);
    }

    Color RGBSlider (Rect screenRect, Color rgb) {
        rgb.r = GUI.HorizontalSlider (screenRect, rgb.r, 0.0f, 1.0f);

        // <- Move the next control down a bit to avoid overlapping
        screenRect.y += 20;
        rgb.g = GUI.HorizontalSlider (screenRect, rgb.g, 0.0f, 1.0f);

        // <- Move the next control down a bit to avoid overlapping
        screenRect.y += 20;

        rgb.b = GUI.HorizontalSlider (screenRect, rgb.b, 0.0f, 1.0f);
        return rgb;
    }
```

```
}
```

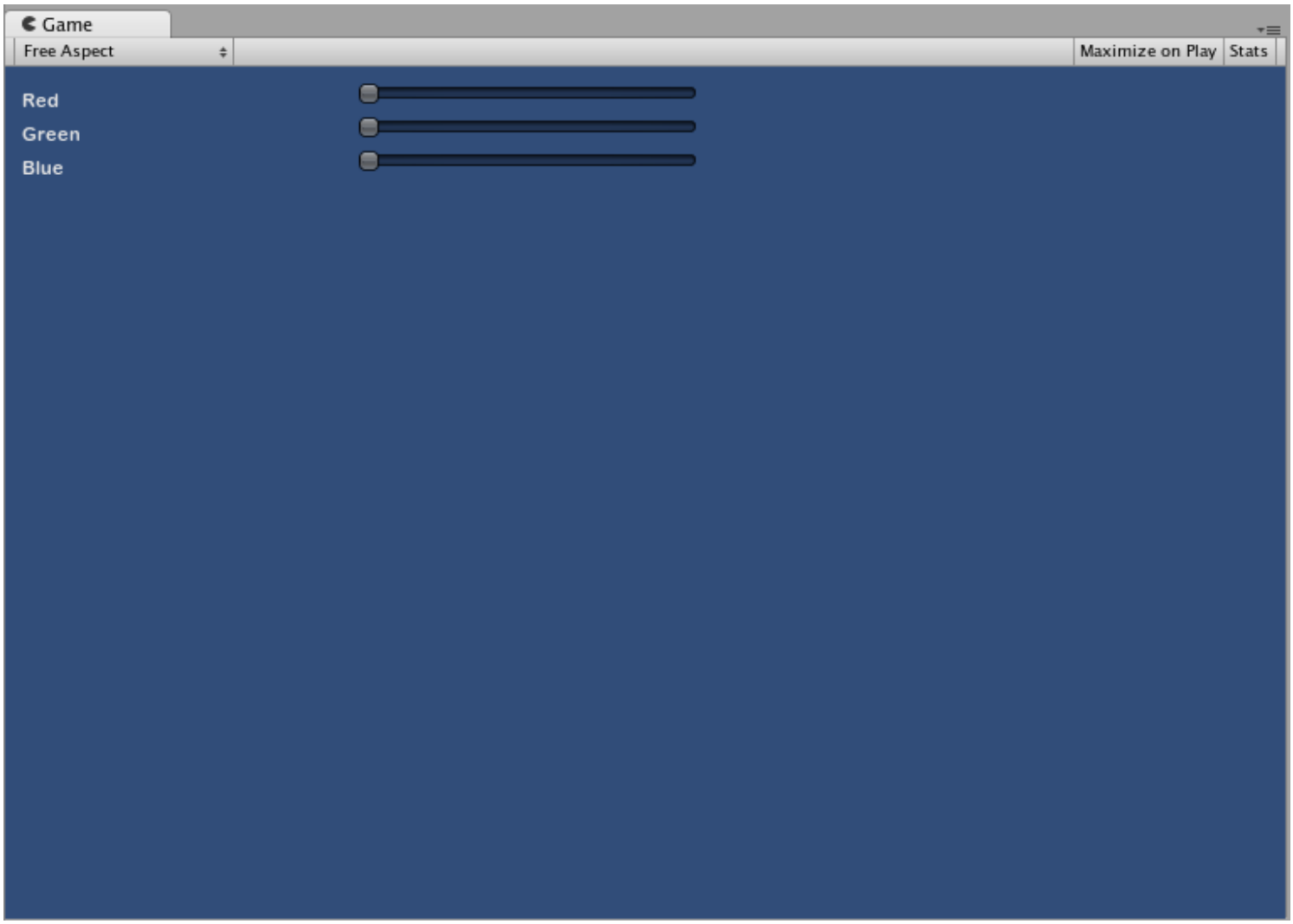


The RGB Slider created by the example above

Now let's build Compound Controls on top of each other, in order to demonstrate how Compound Controls can be used within other Compound Controls. To do this, we will create a new RGB Slider like the one above, but we will use the LabelSlider to do so. This way we'll always have a Label telling us which slider corresponds to which color.

```
/* RGB Label Slider Compound Control */


// JavaScript
var myColor : Color;

function OnGUI () {
    myColor = RGBLabelSlider (Rect (10,10,200,20), myColor);
```

```
}

function RGBLabelSlider (screenRect : Rect, rgb : Color) : Color {
    rgb.r = CompoundControls.LabelSlider (screenRect, rgb.r, 1.0, "Red");
    screenRect.y += 20; // <- Move the next control down a bit to avoid overlapp
    rgb.g = CompoundControls.LabelSlider (screenRect, rgb.g, 1.0, "Green");
    screenRect.y += 20; // <- Move the next control down a bit to avoid overlapp
    rgb.b = CompoundControls.LabelSlider (screenRect, rgb.b, 1.0, "Blue");
    return rgb;
}


// C#
using UnityEngine;
using System.Collections;

public class GUITest : MonoBehaviour {

    public Color myColor;

    void OnGUI () {
        myColor = RGBSlider (new Rect (10,10,200,30), myColor);
    }

    Color RGBSlider (Rect screenRect, Color rgb) {
        rgb.r = CompoundControls.LabelSlider (screenRect, rgb.r, 1.0f, "Red");

        // <- Move the next control down a bit to avoid overlapping
        screenRect.y += 20;
        rgb.g = CompoundControls.LabelSlider (screenRect, rgb.g, 1.0f, "Green");

        // <- Move the next control down a bit to avoid overlapping
        screenRect.y += 20;

        rgb.b = CompoundControls.LabelSlider (screenRect, rgb.b, 1.0f, "Blue");

        return rgb;
    }

}
```



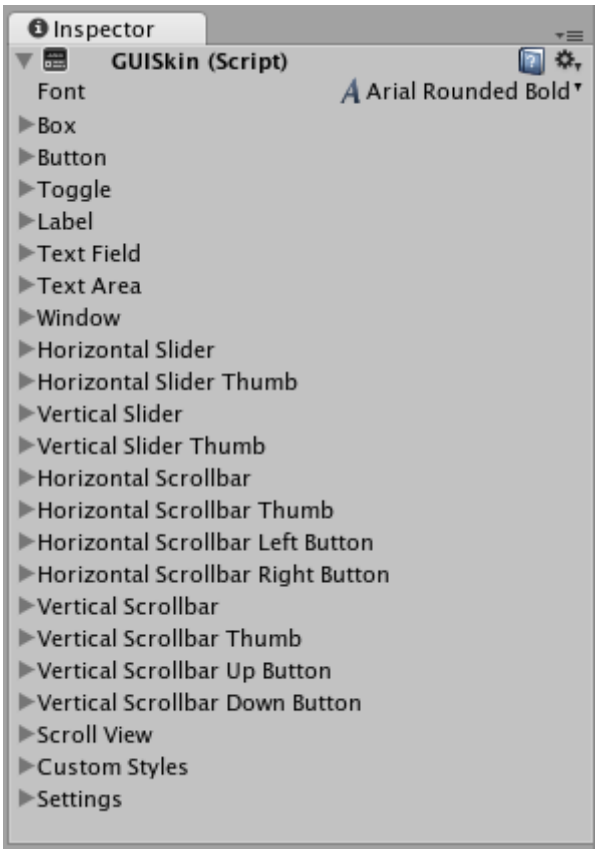
The Compound RGB Label Slider created by the above code

GUI Skin (IMGUI System)

Leave feedback

SWITCH TO SCRIPTING

GUISkins are a collection of **GUIStyles** that can be applied to your GUI. Each **Control** type has its own Style definition. Skins are intended to allow you to apply style to an entire **UI**, instead of a single Control by itself.



A GUI Skin as seen in the Inspector

To create a GUISkin, select **Assets->Create->GUI Skin** from the menubar.

Please Note: This page refers to part of the IMGUI system, which is a *scripting-only* UI system. Unity has a full GameObject-based UI system which you may prefer to use. It allows you to design and edit user interface elements as visible objects in the **scene view**. See the UI System Manual for more information.

Properties

All of the properties within a GUI Skin are an individual GUIStyle. Please read the GUIStyle page for more information about how to use Styles.

Property:	*Function:*
Font	The global Font to use for every Control in the GUI
Box	The Style to use for all Boxes
Button	The Style to use for all Buttons
Toggle	The Style to use for all Toggles
Label	The Style to use for all Labels
Text Field	The Style to use for all Text Fields

Property:	*Function:*
Text Area	The Style to use for all Text Areas
Window	The Style to use for all Windows
Horizontal Slider	The Style to use for all Horizontal Slider bars
Horizontal Slider Thumb	The Style to use for all Horizontal Slider Thumb Buttons
Vertical Slider	The Style to use for all Vertical Slider bars
Vertical Slider Thumb	The Style to use for all Vertical Slider Thumb Buttons
Horizontal Scrollbar	The Style to use for all Horizontal Scrollbars
Horizontal Scrollbar Thumb	The Style to use for all Horizontal Scrollbar Thumb Buttons
Horizontal Scrollbar Left Button	The Style to use for all Horizontal Scrollbar scroll Left Buttons
Horizontal Scrollbar Right Button	The Style to use for all Horizontal Scrollbar scroll Right Buttons
Vertical Scrollbar	The Style to use for all Vertical Scrollbars
Vertical Scrollbar Thumb	The Style to use for all Vertical Scrollbar Thumb Buttons
Vertical Scrollbar Up Button	The Style to use for all Vertical Scrollbar scroll Up Buttons
Vertical Scrollbar Down Button	The Style to use for all Vertical Scrollbar scroll Down Buttons
Custom 1–20	Additional custom Styles that can be applied to any Control
Custom Styles	An array of additional custom Styles that can be applied to any Control
Settings	Additional Settings for the entire GUI
Double Click Selects Word	If enabled, double-clicking a word will select it
Triple Click Selects Line	If enabled, triple-clicking a word will select the entire line
Cursor Color	Color of the keyboard cursor
Cursor Flash Speed	The speed at which the text cursor will flash when editing any Text Control
Selection Color	Color of the selected area of Text

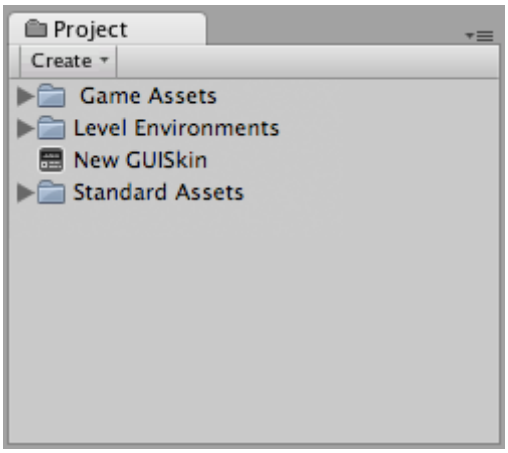
Details

When you are creating an entire GUI for your game, you will likely need to do a lot of customization for every different Control type. In many different game genres, like real-time strategy or role-playing, there is a need for practically every single Control type.

Because each individual Control uses a particular Style, it does not make sense to create a dozen-plus individual Styles and assign them all manually. GUI Skins take care of this problem for you. By creating a GUI Skin, you have a pre-defined collection of Styles for every individual Control. You then apply the Skin with a single line of code, which eliminates the need to manually specify the Style of each individual Control.

Creating GUISkins

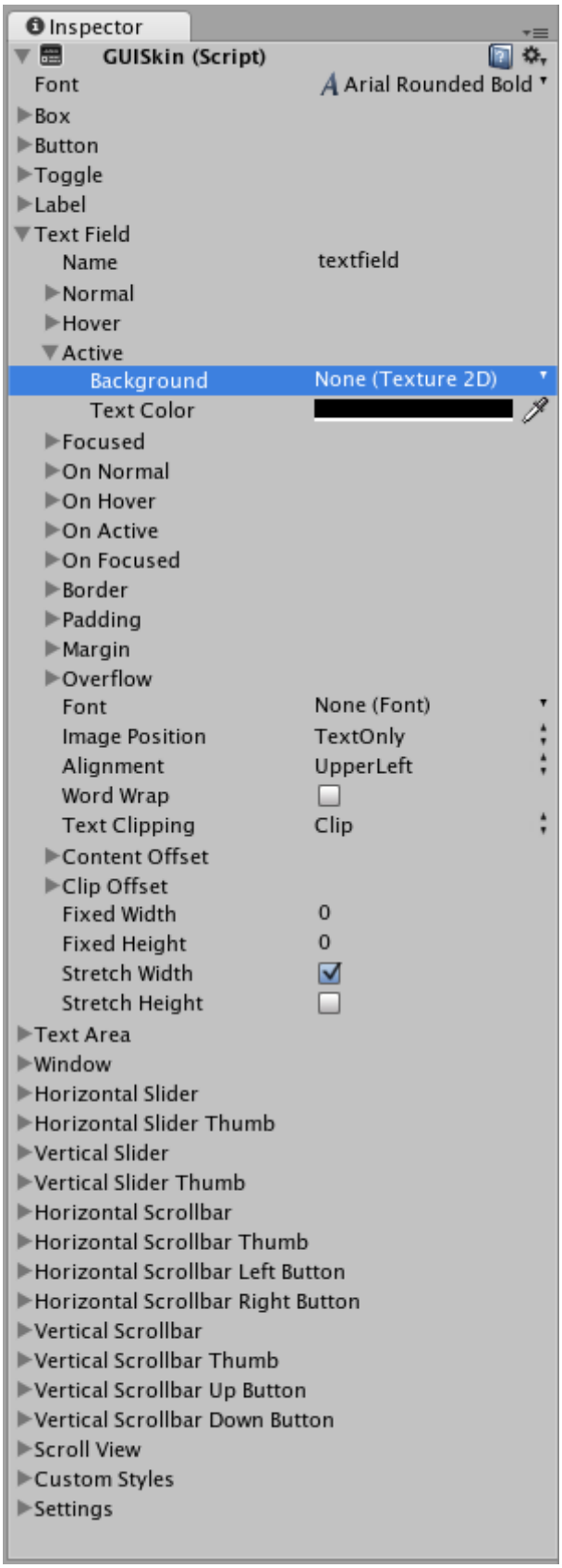
GUISkins are asset files. To create a GUI Skin, select **Assets->Create->GUI Skin** from the menubar. This will put a new GUISkin in your **Project View**.



A new GUISkin file in the Project View

Editing GUISkins

After you have created a GUISkin, you can edit all of the Styles it contains in the **Inspector**. For example, the **Text Field** Style will be applied to all Text Field Controls.



Editing the Text Field Style in a GUISkin

No matter how many Text Fields you create in your script, they will all use this Style. Of course, you have control over changing the styles of one Text Field over the other if you wish. We'll discuss how that is done next.

Applying GUISkins

To apply a GUISkin to your GUI, you must use a simple script to read and apply the Skin to your Controls.

```
// Create a public variable where we can assign the GUISkin
var customSkin : GUISkin;

// Apply the Skin in our OnGUI() function
function OnGUI () {
    GUI.skin = customSkin;

    // Now create any Controls you like, and they will be displayed with the
    GUILayout.Button ("I am a re-Skinned Button");

    // You can change or remove the skin for some Controls but not others
    GUI.skin = null;

    // Any Controls created here will use the default Skin and not the custo
    GUILayout.Button ("This Button uses the default UnityGUI Skin");
}
```

In some cases you want to have two of the same Control with different Styles. For this, it does not make sense to create a new Skin and re-assign it. Instead, you use one of the **Custom** Styles in the skin. Provide a **Name** for the custom Style, and you can use that name as the last argument of the individual Control.

```
// One of the custom Styles in this Skin has the name "MyCustomControl"
var customSkin : GUISkin;

function OnGUI () {
    GUI.skin = customSkin;

    // We provide the name of the Style we want to use as the last argument
    GUILayout.Button ("I am a custom styled Button", "MyCustomControl");

    // We can also ignore the Custom Style, and use the Skin's default Butto
    GUILayout.Button ("I am the Skin's Button Style");
}
```

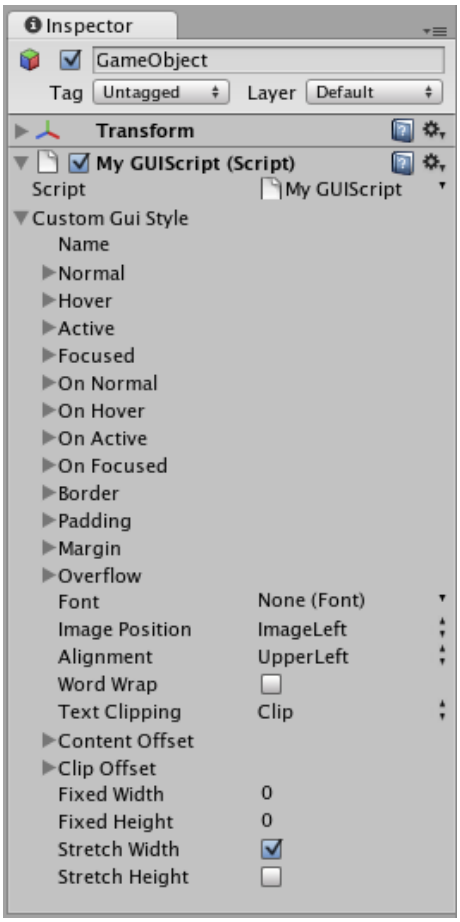
For more information about working with GUIStyles, please read the GUIStyle page. For more information about using UnityGUI, please read the GUI Scripting Guide.

GUI Style (IMGUI System)

Leave feedback

SWITCH TO SCRIPTING

GUI Styles are a collection of custom attributes for use with **UnityGUI**. A single GUI Style defines the appearance of a single UnityGUI **Control**.



A GUI Style in the Inspector

If you want to add style to more than one control, use a GUI Skin instead of a GUI Style. For more information about UnityGUI, please read the GUI Scripting Guide.

Please Note: This page refers to part of the IMGUI system, which is a *scripting-only* **UI** system. Unity has a full GameObject-based **UI** system which you may prefer to use. It allows you to design and edit user interface elements as visible objects in the **scene view**. See the UI System Manual for more information.

Properties

Property:	*Function:*
Name	The text string that can be used to refer to this specific Style
Normal	Background image & Text Color of the Control in default state
Hover	Background image & Text Color when the mouse is positioned over the Control
Active	Background image & Text Color when the mouse is actively clicking the Control
Focused	Background image & Text Color when the Control has keyboard focus
On Normal	Background image & Text Color of the Control in enabled state
On Hover	Background image & Text Color when the mouse is positioned over the enabled Control
On Active	Properties when the mouse is actively clicking the enabled Control
On Focused	Background image & Text Color when the enabled Control has keyboard focus

Property:	*Function:*
Border	Number of **pixels** on each side of the **Background** image that are not affected by the scale of the Control' shape
Padding	Space in pixels from each edge of the Control to the start of its contents.
Margin	The margins between elements rendered in this style and any other GUI Controls.
Overflow	Extra space to be added to the background image.
Font	The Font used for all text in this style
Image Position	The way the background image and text are combined.
Alignment	Standard text alignment options.
Word Wrap	If enabled, text that reaches the boundaries of the Control will wrap around to the next line
Text Clipping	If **Word Wrap** is enabled, choose how to handle text that exceeds the boundaries of the Control
Overflow	Any text that exceeds the Control boundaries will continue beyond the boundaries
Clip	Any text that exceeds the Control boundaries will be hidden
Content Offset	Number of pixels along X and Y axes that the Content will be displaced in addition to all other properties
X	Left/Right Offset
Y	Up/Down Offset
Fixed Width	Number of pixels for the width of the Control, which will override any provided **Rect()** value
Fixed Height	Number of pixels for the height of the Control, which will override any provided **Rect()** value
Stretch Width	If enabled, Controls using this style can be stretched horizontally for a better layout.
Stretch Height	If enabled, Controls using this style can be stretched vertically for a better layout.

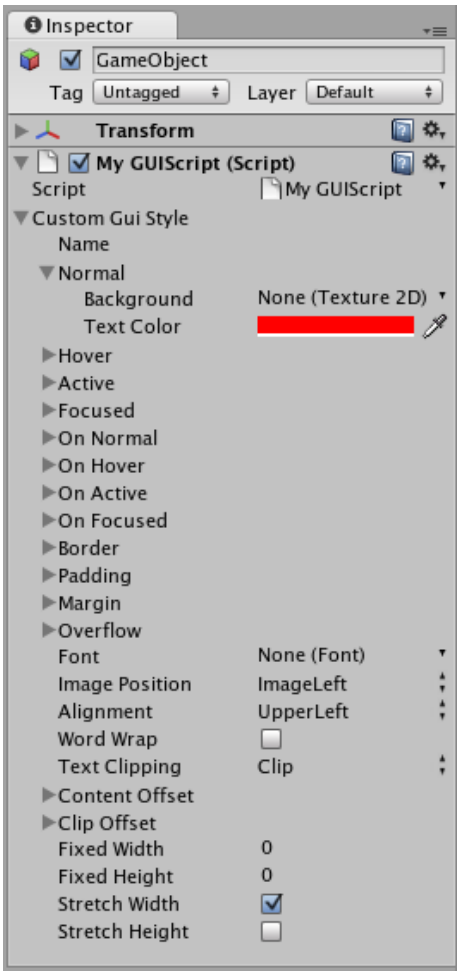
Details

GUIStyles are declared from **scripts** and modified on a per-instance basis. If you want to use a single or few Controls with a custom Style, you can declare this custom Style in the script and provide the Style as an argument of the Control function. This will make these Controls appear with the Style that you define.

First, you must declare a GUI Style from within a script.

```
/* Declare a GUI Style */
var customGuiStyle : GUIStyle;

...
```

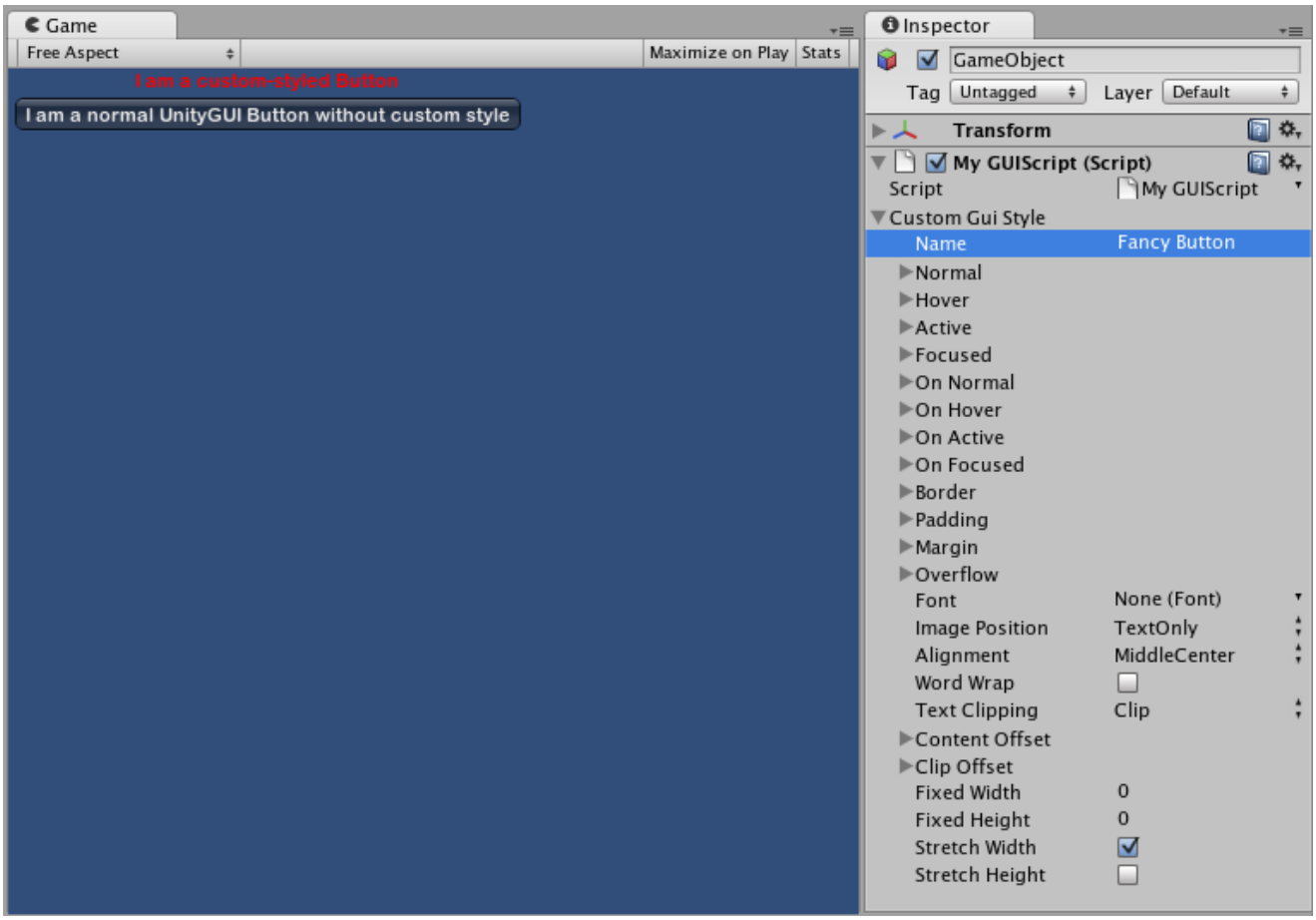
When you attach this script to a **GameObject**, you will see the custom Style available to modify in the **Inspector**.



A Style declared in a script can be modified in each instance of the script

Now, when you want to tell a particular Control to use this Style, you provide the name of the Style as the last argument in the Control function.

```
...

function OnGUI () {
    // Provide the name of the Style as the final argument to use it
    GUILayout.Button ("I am a custom-styled Button", customGuiStyle);

    // If you do not want to apply the Style, do not provide the name
    GUILayout.Button ("I am a normal UnityGUI Button without custom style");
}
```

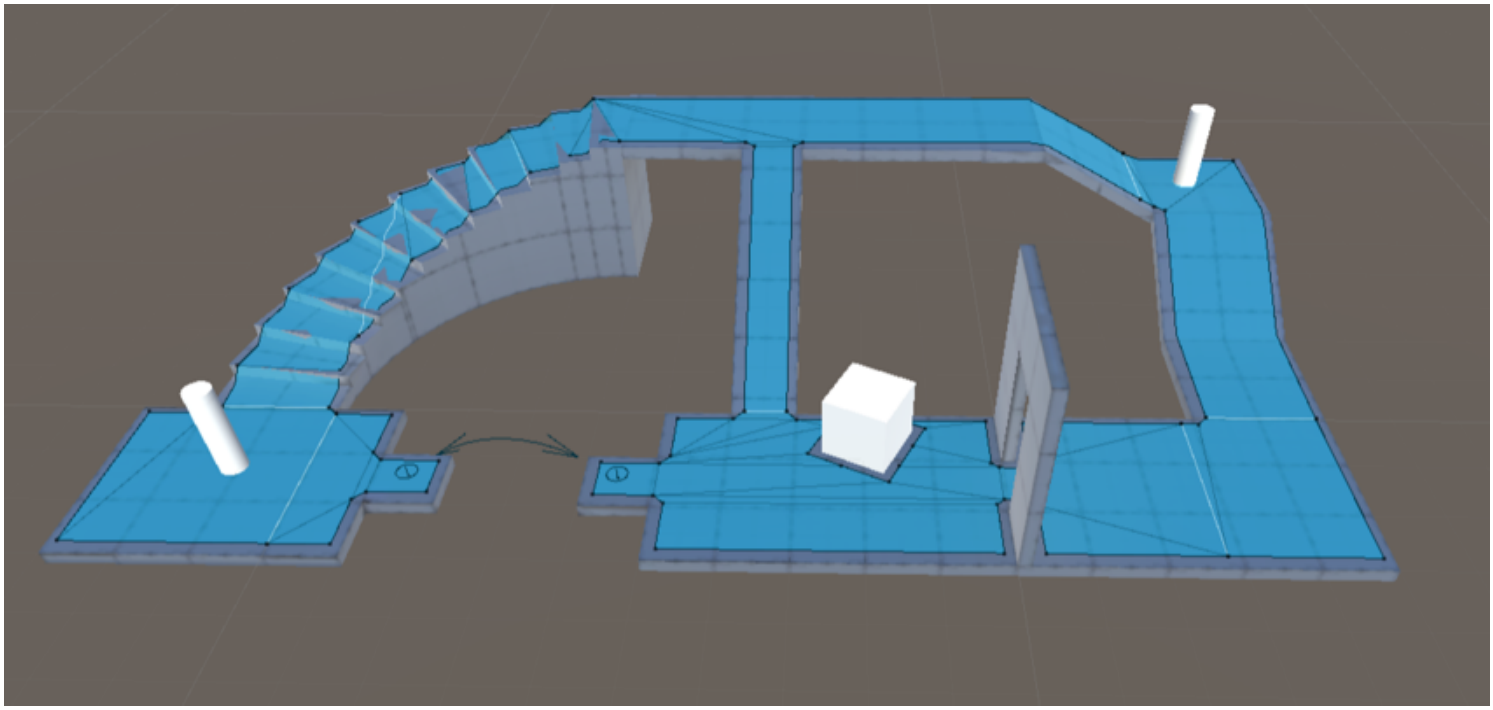


Two Buttons, one with Style, as created by the code example

For more information about using UnityGUI, please read the GUI Scripting Guide.

Navigation and Pathfinding

Leave feedback



The navigation system allows you to create characters that can intelligently move around the game world, using navigation meshes that are created automatically from your **Scene** geometry. Dynamic obstacles allow you to alter the navigation of the characters at runtime, while off-mesh links let you build specific actions like opening doors or jumping down from a ledge. This section describes Unity's navigation and pathfinding systems in detail.

Related tutorials: Navigation

Search the Unity Knowledge Base for tips, tricks and troubleshooting.

Navigation Overview

Leave feedback

This section will dive into the details on building NavMeshes for your **scene**, creating **NavMesh** Agents, **NavMesh** Obstacles and Off-Mesh Links.

Navigation System in Unity

Leave feedback

The Navigation System allows you to create characters which can navigate the game world. It gives your characters the ability to understand that they need to take stairs to reach second floor, or to jump to get over a ditch. The Unity **NavMesh** system consists of the following pieces:

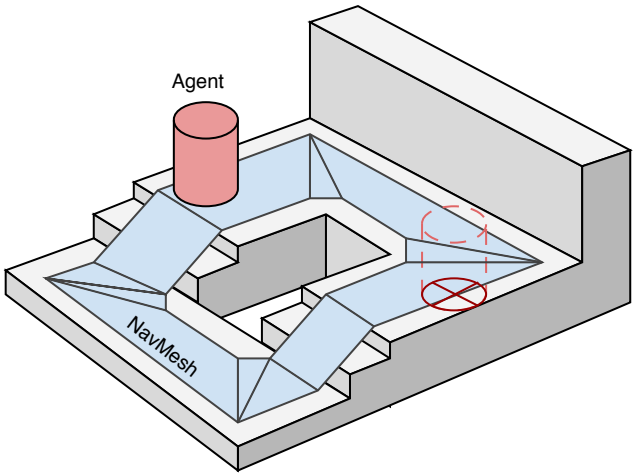
NavMesh (short for Navigation Mesh) is a data structure which describes the walkable surfaces of the game world and allows to find path from one walkable location to another in the game world. The data structure is built, or baked, automatically from your level geometry.
NavMesh Agent component help you to create characters which avoid each other while moving towards their goal. Agents reason about the game world using the NavMesh and they know how to avoid each other as well as moving obstacles.
Off-Mesh Link component allows you to incorporate navigation shortcuts which cannot be represented using a walkable surface. For example, jumping over a ditch or a fence, or opening a door before walking through it, can be all described as Off-mesh links.
NavMesh Obstacle component allows you to describe moving obstacles the agents should avoid while navigating the world. A barrel or a crate controlled by the physics system is a good example of an obstacle. While the obstacle is moving the agents do their best to avoid it, but once the obstacle becomes stationary it will carve a hole in the navmesh so that the agents can change their paths to steer around it, or if the stationary obstacle is blocking the path way, the agents can find a different route.

Inner Workings of the Navigation System

Leave feedback

When you want to intelligently move characters in your game (or agents as they are called in AI circles), you have to solve two problems: how to *reason* about the level to find the destination, then how to *move* there. These two problems are tightly coupled, but quite different in nature. The problem of reasoning about the level is more global and static, in that it takes into account the whole **scene**. Moving to the destination is more local and dynamic, it only considers the direction to move and how to prevent **collisions** with other moving agents.

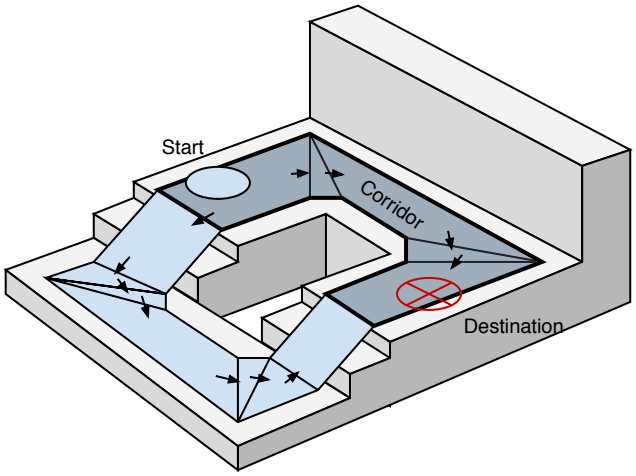
Walkable Areas



The navigation system needs its own data to represent the walkable areas in a game scene. The walkable areas define the places in the scene where the agent can stand and move. In Unity the agents are described as cylinders. The walkable area is built automatically from the geometry in the scene by testing the locations where the agent can stand. Then the locations are connected to a surface laying on top of the scene geometry. This surface is called the navigation **mesh** (NavMesh for short).

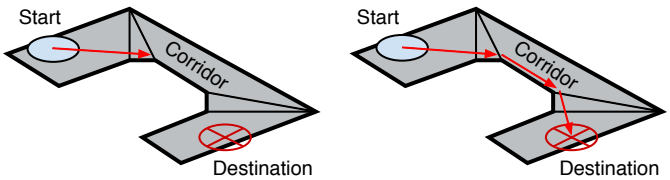
The **NavMesh** stores this surface as convex polygons. Convex polygons are a useful representation, since we know that there are no obstructions between any two points inside a polygon. In addition to the polygon boundaries, we store information about which polygons are neighbours to each other. This allows us to reason about the whole walkable area.

Finding Paths



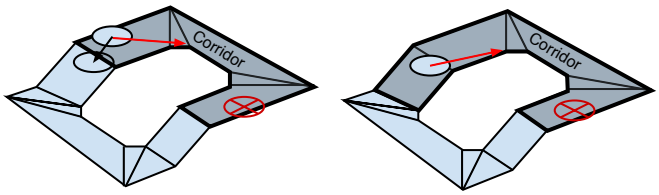
To find path between two locations in the scene, we first need to map the start and destination locations to their nearest polygons. Then we start searching from the start location, visiting all the neighbours until we reach the destination polygon. Tracing the visited polygons allows us to find the sequence of polygons which will lead from the start to the destination. A common algorithm to find the path is A* (pronounced “A star”), which is what Unity uses.

Following the Path



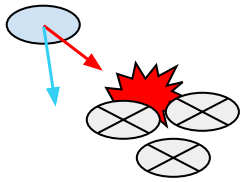
The sequence of polygons which describe the path from the start to the destination polygon is called a corridor. The agent will reach the destination by always steering towards the next visible corner of the corridor. If you have a simple game where only one agent moves in the scene, it is fine to find all the corners of the corridor in one swoop and animate the character to move along the line segments connecting the corners.

When dealing with multiple agents moving at the same time, they will need to deviate from the original path when avoiding each other. Trying to correct such deviations using a path consisting of line segments soon becomes very difficult and error prone.



Since the agent movement in each frame is quite small, we can use the connectivity of the polygons to fix up the corridor in case we need to take a little detour. Then we quickly find the next visible corner to steer towards.

Avoiding Obstacles



The steering logic takes the position of the next corner and based on that figures out a desired direction and speed (or velocity) needed to reach the destination. Using the desired velocity to move the agent can lead to collision with other agents.

Obstacle avoidance chooses a new velocity which balances between moving in the desired direction and preventing future collisions with other agents and edges of the navigation mesh. Unity is using reciprocal velocity obstacles (RVO) to predict and prevent collisions.

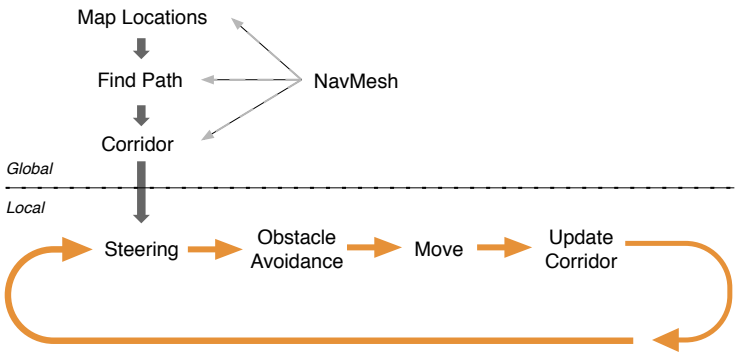
Moving the Agent

Finally after steering and obstacle avoidance the final velocity is calculated. In Unity the agents are simulated using a simple dynamic model, which also takes into account acceleration to allow more natural and smooth movement.

At this stage you can feed the velocity from the simulated agent to the animation system to move the character using **root motion**, or let the navigation system take care of that.

Once the agent has been moved using either method, the simulated agent location is moved and constrained to NavMesh. This last small step is important for robust navigation.

Global and Local



One of the most important things to understand about navigation is the difference between global and local navigation.

Global navigation is used to find the corridor across the world. Finding a path across the world is a costly operation requiring quite a lot of processing power and memory.

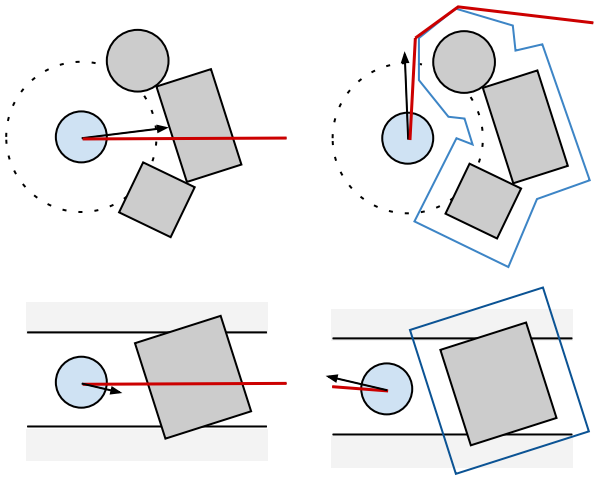
The linear list of polygons describing the path is a flexible data structure for steering, and it can be locally adjusted as the agent's position moves. Local navigation tries to figure out how to efficiently move towards the next corner without colliding with other agents or moving objects.

Two Cases for Obstacles

Many applications of navigation require other types of obstacles rather than just other agents. These could be the usual crates and barrels in a shooter game, or vehicles. The obstacles can be handled using local obstacle avoidance or global pathfinding.

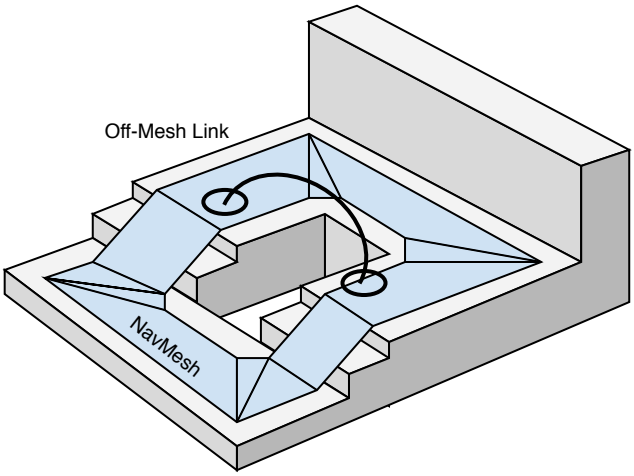
When an obstacle is moving, it is best handled using local obstacles avoidance. This way the agent can predictively avoid the obstacle. When the obstacle becomes stationary, and can be considered to block the path of all agents, the obstacles should affect the global navigation, that is, the navigation mesh.

Changing the NavMesh is called carving. The process detects which parts of the obstacle touches the NavMesh and carves holes into the NavMesh. This is computationally expensive operation, which is yet another compelling reason, why moving obstacles should be handled using collision avoidance.



Local collision avoidance can be often used to steer around sparsely scattered obstacles too. Since the algorithm is local, it will only consider the next immediate collisions, and cannot steer around traps or handle cases where the an obstacle blocks a path. These cases can be solved using carving.

Describing Off-mesh Links



The connections between the NavMesh polygons are described using links inside the pathfinding system. Sometimes it is necessary to let the agent to navigate across places which are not walkable, for example, jumping over a fence, or traversing through a closed door. These cases need to know the location of the action.

These actions can be annotated using Off-Mesh Links which tell the pathfinder that a route exists through the specified link. This link can be later accessed when following the path, and the special action can be executed.

Building a NavMesh

Leave feedback

The process of creating a **NavMesh** from the level geometry is called **NavMesh** Baking. The process collects the Render Meshes and **Terrains** of all Game Objects which are marked as Navigation Static, and then processes them to create a navigation **mesh** that approximates the walkable surfaces of the level.

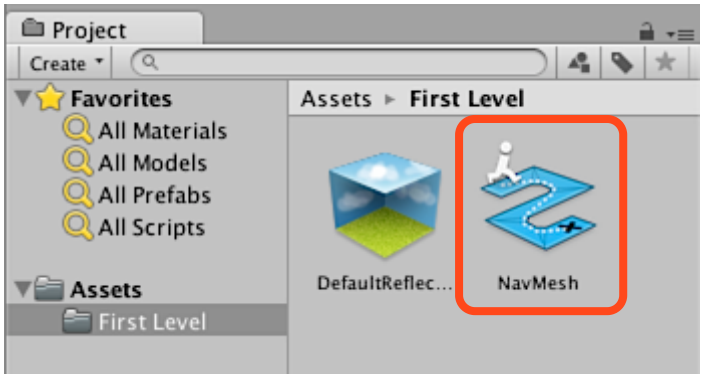
In Unity, NavMesh generation is handled from the Navigation window (menu: **Window** > **AI** > **Navigation**).

Building a NavMesh for your **scene** can be done in 4 quick steps:

Select scene geometry that should affect the navigation – walkable surfaces and obstacles.
Check Navigation Static on to include selected objects in the NavMesh baking process.
Adjust the bake settings to match your agent size.
Agent Radius defines how close the agent center can get to a wall or a ledge.
Agent Height defines how low the spaces are that the agent can reach.
Max Slope defines how steep the ramps are that the agent walk up.
Step Height defines how high obstructions are that the agent can step on.
Click bake to build the NavMesh.
The resulting NavMesh will be shown in the scene as a blue overlay on the underlying level geometry whenever the Navigation Window is open and visible.

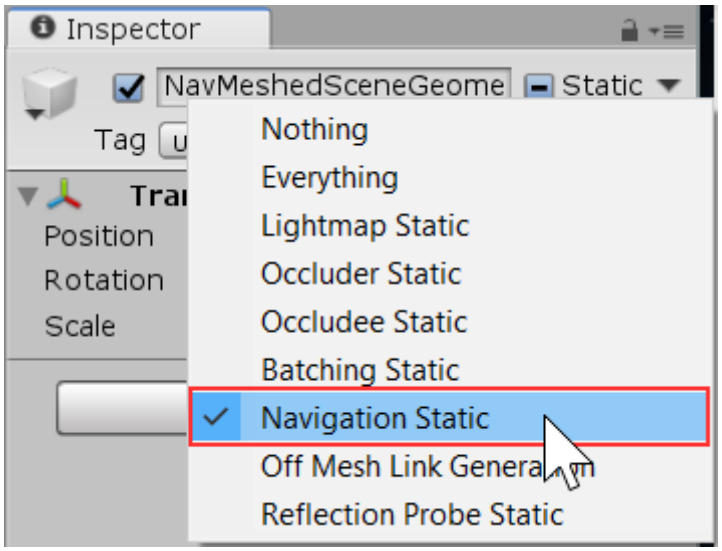
As you may have noticed in the above pictures, the walkable area in the generated NavMesh appears shrunk. The NavMesh represents the area where the center of the agent can move. Conceptually, it doesn't matter whether you regard the agent as a point on a shrunken NavMesh or a circle on a full-size NavMesh since the two are equivalent. However, the point interpretation allows for better runtime efficiency and also allows the designer to see immediately whether an agent can squeeze through gaps without worrying about its radius.

Another thing to keep in mind is that the NavMesh is an approximation of the walkable surface. This can be seen for example in the stairs which are represented as a flat surface, while the source surface has steps. This is done in order to keep the NavMesh data size small. The side effect of the approximation is that sometimes you will need to have a little extra space in your level geometry to allows the agent to pass through a tight spot.



When baking is complete, you will find a NavMesh asset file inside a folder with the same name as the scene the NavMesh belongs to. For example, if you have a scene called *First Level* in the Assets folder, the NavMesh will be at *Assets > First Level > NavMesh.asset*.

Additional Workflows for Marking Objects for Baking



In addition to marking objects as *Navigation Static* in the Navigation Window, as explained above, you can use the *Static* menu at the top of the **inspector**. This can be convenient if you don't happen to have the Navigation Window open.

Further Reading

Creating a NavMeshAgent – learn how to allow your characters to move.
Bake Settings – full description of the NavMesh bake settings.
Areas and Costs – learn how to use different Area types.

NavMesh building components

Leave feedback

NavMesh building components provide you with additional controls for building (also known as baking) and using NavMeshes at run time and in the Unity Editor.

The high-level NavMesh building components listed below are not supplied with the standard Unity Editor installer which you download from the Unity store. Instead, download them from the Unity Technologies GitHub and install them separately.

There are four high-level components to use with NavMesh:

NavMesh Surface - Use for building and enabling a NavMesh surface for one type of Agent.

NavMesh Modifier - Use for affecting the NavMesh generation of NavMesh area types based on the transform hierarchy.

NavMeshModifierVolume - Use for affecting the NavMesh generation of NavMesh area types based on volume.

NavMeshLink - Use for connecting the same or different NavMesh surfaces for one type of Agent.

See also documentation on Mesh-BuildingComponents-API.

For more information on Agent types, refer to documentation on creating NavMesh Agents.

For more details about NavMesh area types, see documentation on NavMesh areas.

Creating high-level NavMesh building components

To install the high-level NavMesh building components:

Download and install Unity 5.6 or later.

Clone or download the repository from the NavMesh Components page on the Unity Technologies GitHub by clicking on the green **Clone or download** button.

Open the NavMesh Components Project using Unity, or alternatively copy the contents of the *Assets/NavMeshComponents* folder to an existing Project.

Find additional examples in the *Assets/Examples* folder.

Note: Make sure to back up your Project before installing high-level NavMesh building components.

2017–05–26 Page published with limited editorial review

New feature in 5.6

NavMesh Surface

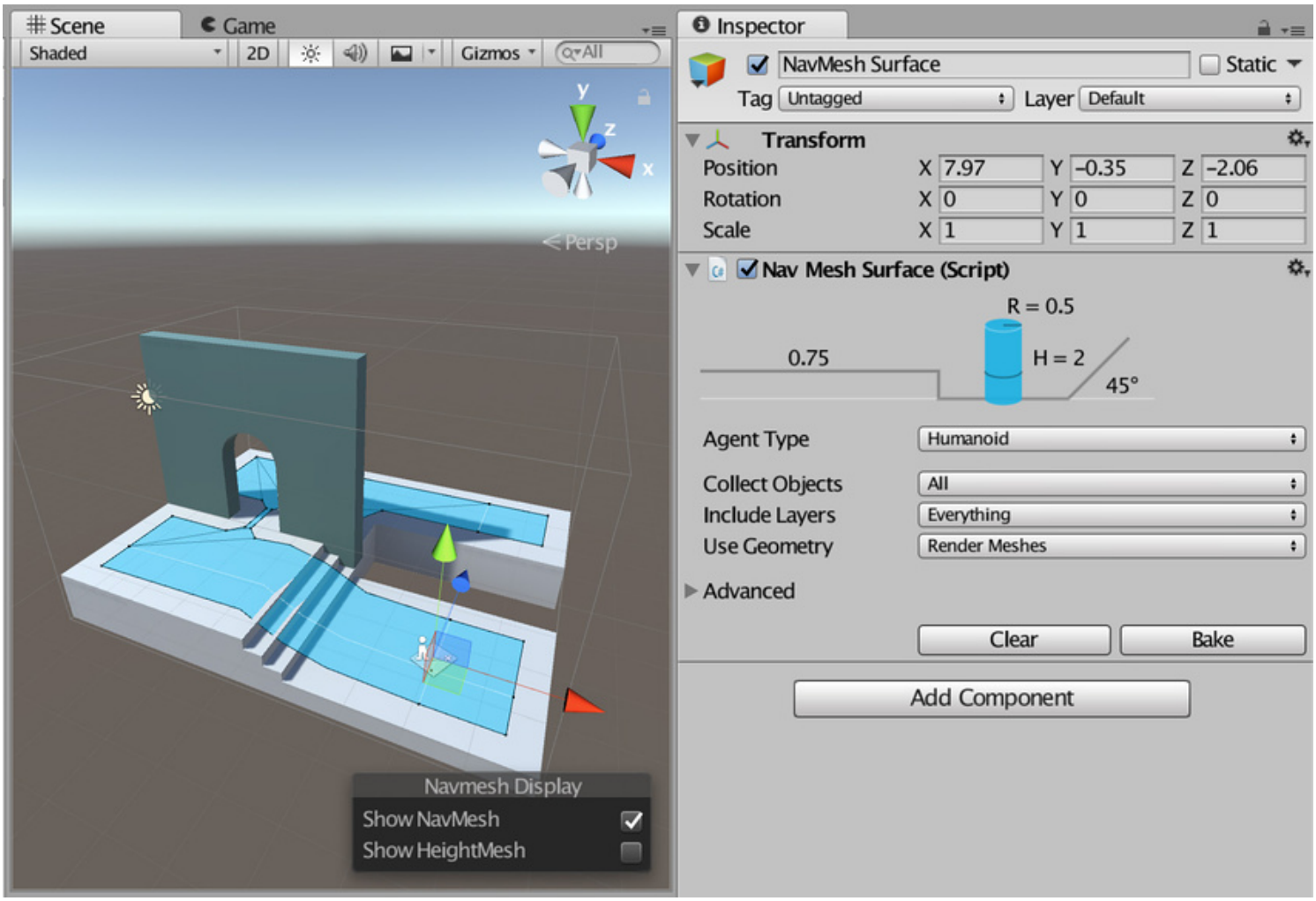
Leave feedback

The **NavMesh** Surface **component** represents the walkable area for a specific NavMesh Agent type, and defines a part of the **Scene** where a NavMesh should be built.

The NavMesh Surface component is not in the standard Unity install; see documentation on high-level NavMesh building components for information on how to access it.

To use the NavMesh Surface component, navigate to **GameObject** > **AI** > **NavMesh Surface**. This creates an empty GameObject with a NavMesh Surface component attached to it. A Scene can contain multiple NavMesh Surfaces.

You can add the NavMesh Surface component to any GameObject. This is useful for when you want to use the GameObject parenting Hierarchy to define which GameObjects contribute to the NavMesh.



A NavMesh Surface component open in the Inspector window

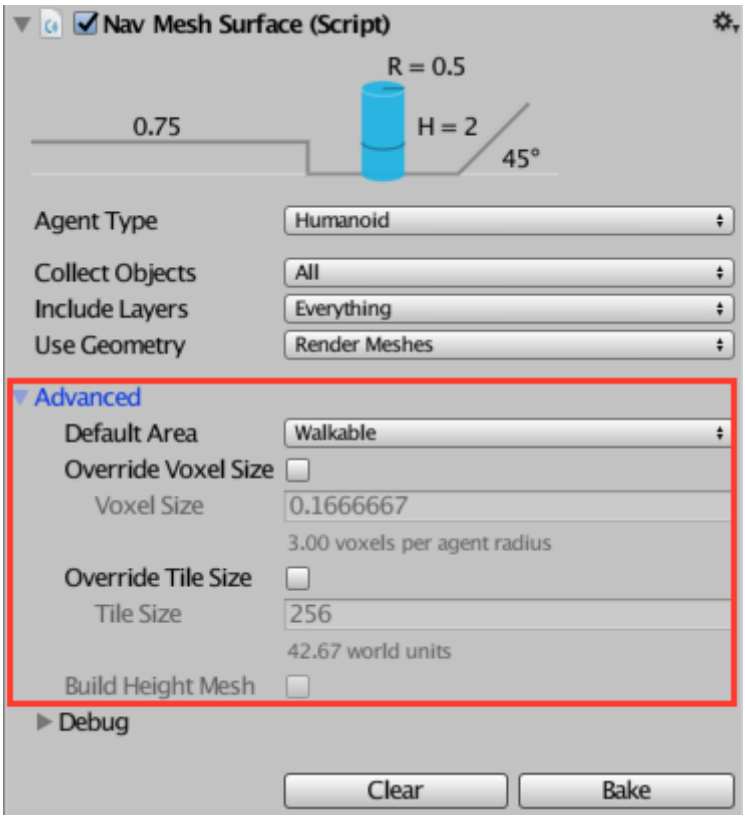
Property	Function
Agent Type	The NavMesh Agent type using the NavMesh Surface. Use for bake settings and matching the NavMesh Agent to proper surfaces during pathfinding. - Humanoid - Ogre

Property	Function
Collect Objects	Defines which GameObjects to use for baking. - **All** – Use all active GameObjects (this is the default option). - **Volume** – Use all active GameObjects overlapping the bounding volume. - **Children** – Use all active GameObjects which are children of the NavMesh Surface component.
Include Layers	Define the layers on which GameObjects are included in the bake. In addition to **Collect Objects**, this allows for further exclusion of specific GameObjects from the bake (for example, effects or animated characters). This is set to **Everything** by default, but you can toggle following options on (denoted by a tick) or off individually: - **Nothing** (automatically unticks all other options, turning them off) - **Everything** (automatically ticks all other options, turning them on) - **Default** - **TransparentFX** - **Ignore Raycast** - **Water** - **UI**
Use Geometry	Select which geometry to use for baking. - **Render Meshes** – Use geometry from Render Meshes and Terrains. - **Physics Colliders** – Use geometry from Colliders and Terrains. Agents can move closer to the edge of the physical bounds of the environment with this option than they can with the **Render Meshes** option.

Use the main settings for the NavMesh Surface component to filter the input geometry on a broad scale. Fine tune how Unity treats input geometry on a per-GameObject basis, using the NavMesh Modifier component.

The baking process automatically excludes GameObjects that have a NavMesh Agent or NavMesh Obstacle. They are dynamic users of the NavMesh, and so do not contribute to NavMesh building.

Advanced settings



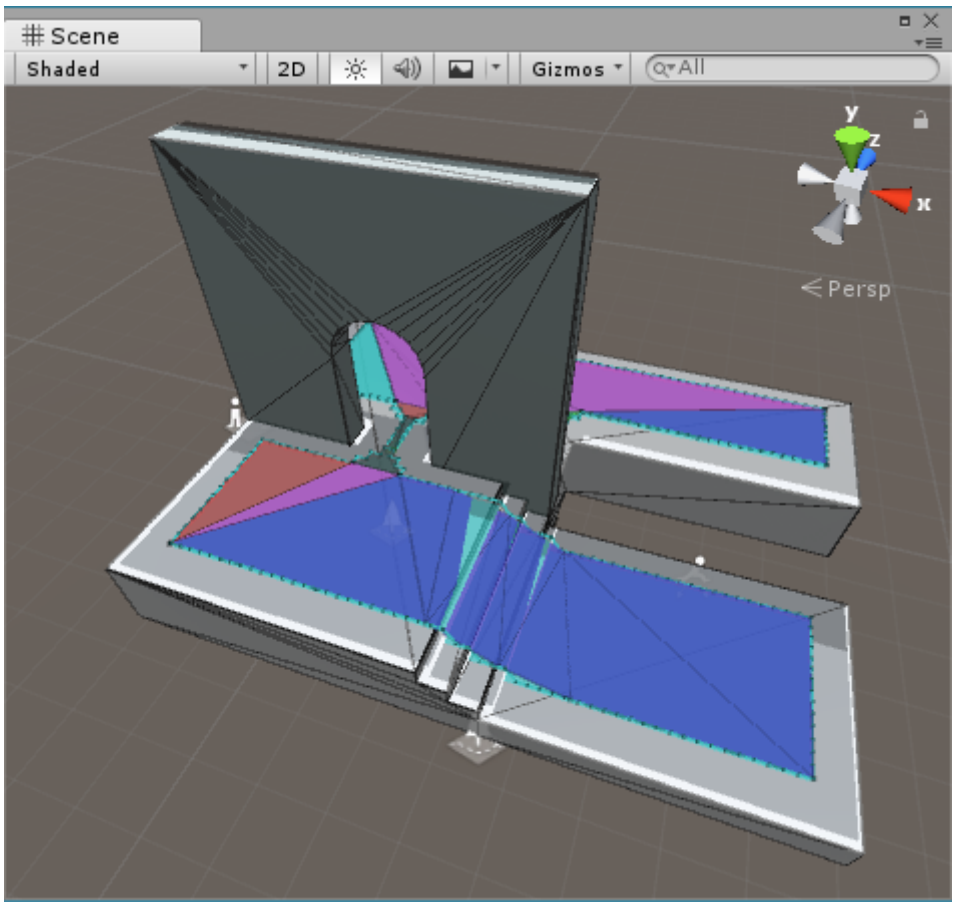
The NavMesh Surface Advanced settings panel

The Advanced settings section allows you to customize the following additional parameters:

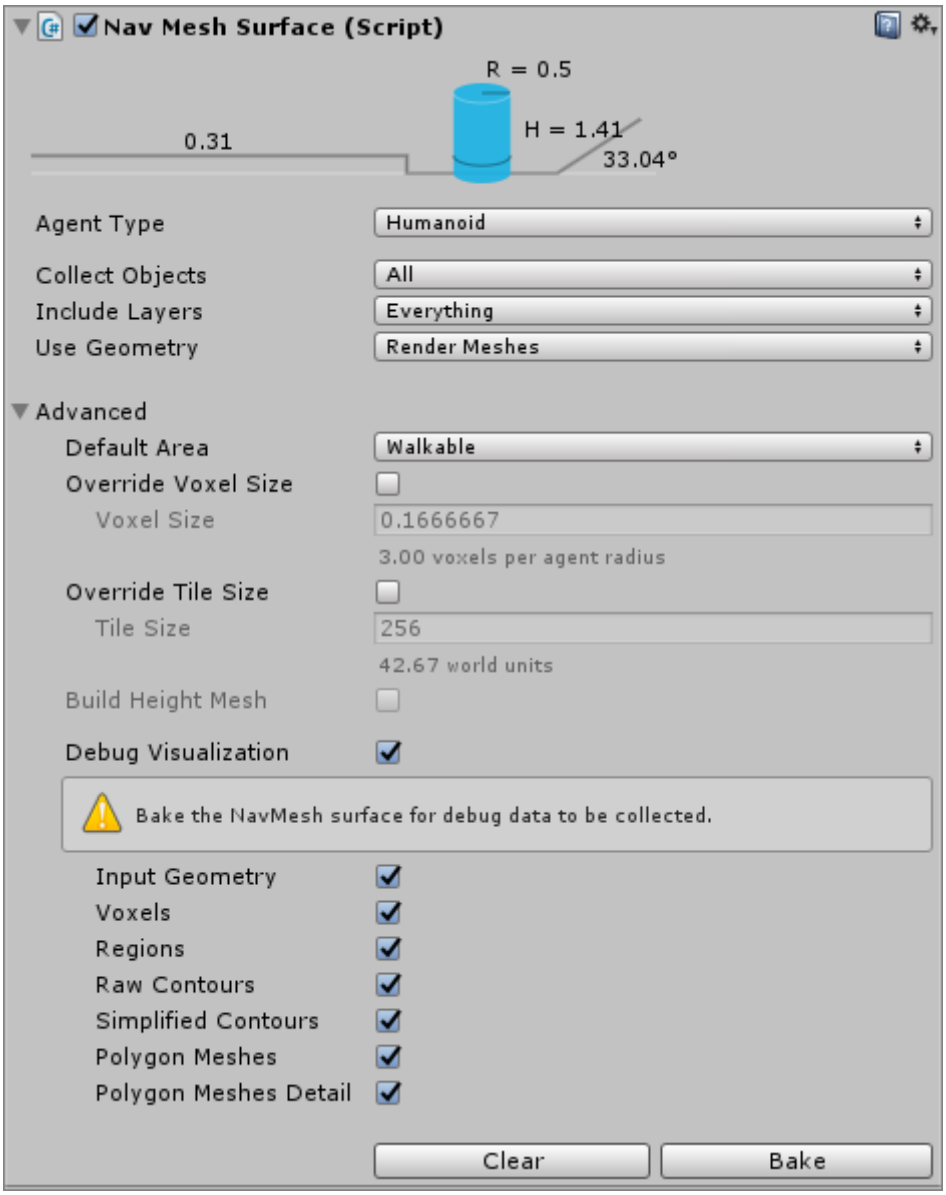
Property	Function
Default Area	Defines the area type generated when building the NavMesh. - **Walkable** (this is the default option) - **Not Walkable** - **Jump** Use the NavMesh Modifier component to modify the area type in more detail.
Override Voxel Size	Controls how accurately Unity processes the input geometry for NavMesh baking (this is a tradeoff inbetween speed and accuracy). Check the tickbox to enable. The default is unchecked (disabled). 3 voxels per Agent radius (6 per diameter) allows the capture of narrow passages, such as doors, while maintaining a quick baking time. For big open areas, using 1 or 2 voxels per radius speeds up baking. Tight indoor spots are better suited to smaller voxels, for example 4 to 6 voxels per radius. More than 8 voxels per radius does not usually provide much additional benefit.

Property	Function
Override Tile Size	In order to make the bake process parallel and memory efficient, the Scene is divided into tiles for baking. The white lines visible on the NavMesh are tile boundaries. The default tile size is 256 voxels, which provides a good tradeoff between memory usage and NavMesh fragmentation. To change this default tile size, check this tickbox and, in the **Tile Size** field, enter the number of voxels you want the Tile Size to be. The smaller the tiles, the more fragmented the NavMesh is. This can sometimes cause non-optimal paths. NavMesh carving also operates on tiles. If you have a lot of obstacles, you can often speed up carving by making the tile size smaller (for example around 64 to 128 voxels). If you plan to bake the NavMesh at runtime, using a smaller tile size to keep the maximum memory usage low.
Build Height Mesh	Not supported.

Advanced Debug Visualization



Input Geometry, Regions, Polygonal Mesh Detail and Raw Contours shown after building the NavMesh with debug options



Nav Mesh Surface Inspector with Debug Visualization options

Use the settings in the **Debug Visualization** section to diagnose any problems encountered during the NavMesh building process. The different tickboxes show each step of the NavMesh building process, including input scene voxelization (**Input Geometry**), region splitting (**Regions**), contour generation (**Contours**) and the NavMesh polygons (**Polygon Meshes**).

2017–09–14 Page published with limited editorial review

New feature in 5.6

New Advanced Debug Visualization added in 2017.2

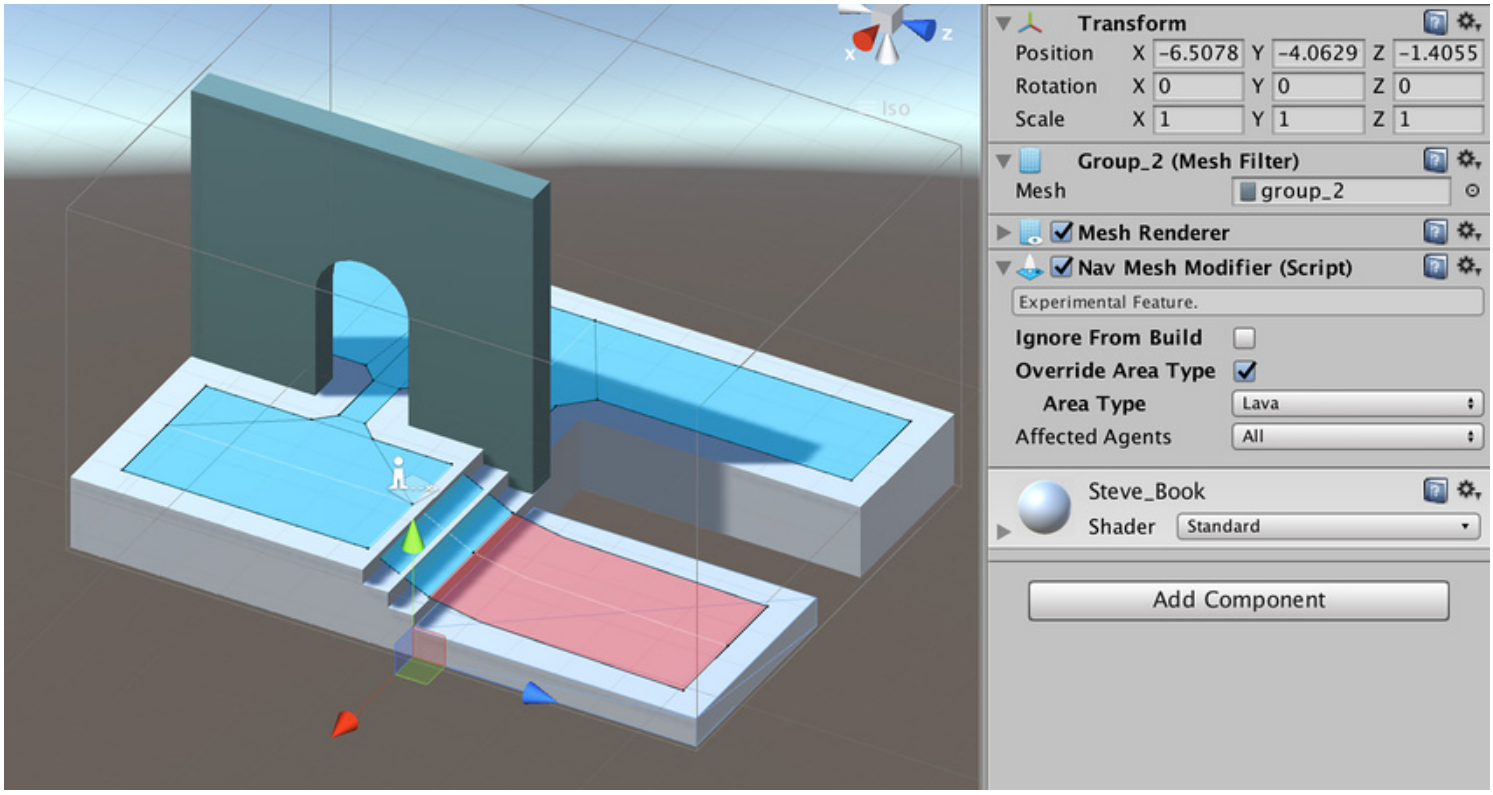
NavMesh Modifier

Leave feedback

NavMesh Modifiers adjust how a specific GameObject behaves during **NavMesh** baking at runtime. **NavMesh** Modifiers are not in the Unity standard install; see documentation on high-level NavMesh building components for information on how to access them.

To use the NavMesh Modifier component, navigate to **GameObject** > **AI** > **NavMesh Modifier**.

In the image below, the platform in the bottom right has a modifier attached to it that sets its **Area Type** to **Lava**.



A NavMesh Modifier component open in the Inspector window

The NavMesh Modifier affects GameObjects hierarchically, meaning the GameObject that the **component** is attached to as well as all its children are affected. Additionally, if another NavMesh Modifier is found further down the transform hierarchy, the new NavMesh Modifier overrides the one further up the hierarchy.

The NavMesh Modifier also affects the NavMesh generation process, meaning the NavMesh has to be updated to reflect any changes to NavMesh Modifiers.

Property:	Function:
Ignore From Build	Check this tickbox to exclude the GameObject and all if its children from the build process.
Override Area Type	Check this tickbox to change the area type for the GameObject containing the Modifier and all of its children.
Area Type	Select the new area type to apply from the drop-down menu.
Affected Agents	A selection of Agents the Modifier affects. For example, you may choose to exclude certain obstacles from specific Agents.

2017–05–26 Page published with limited editorial review

New feature in 5.6

NavMesh Modifier Volume

Leave feedback

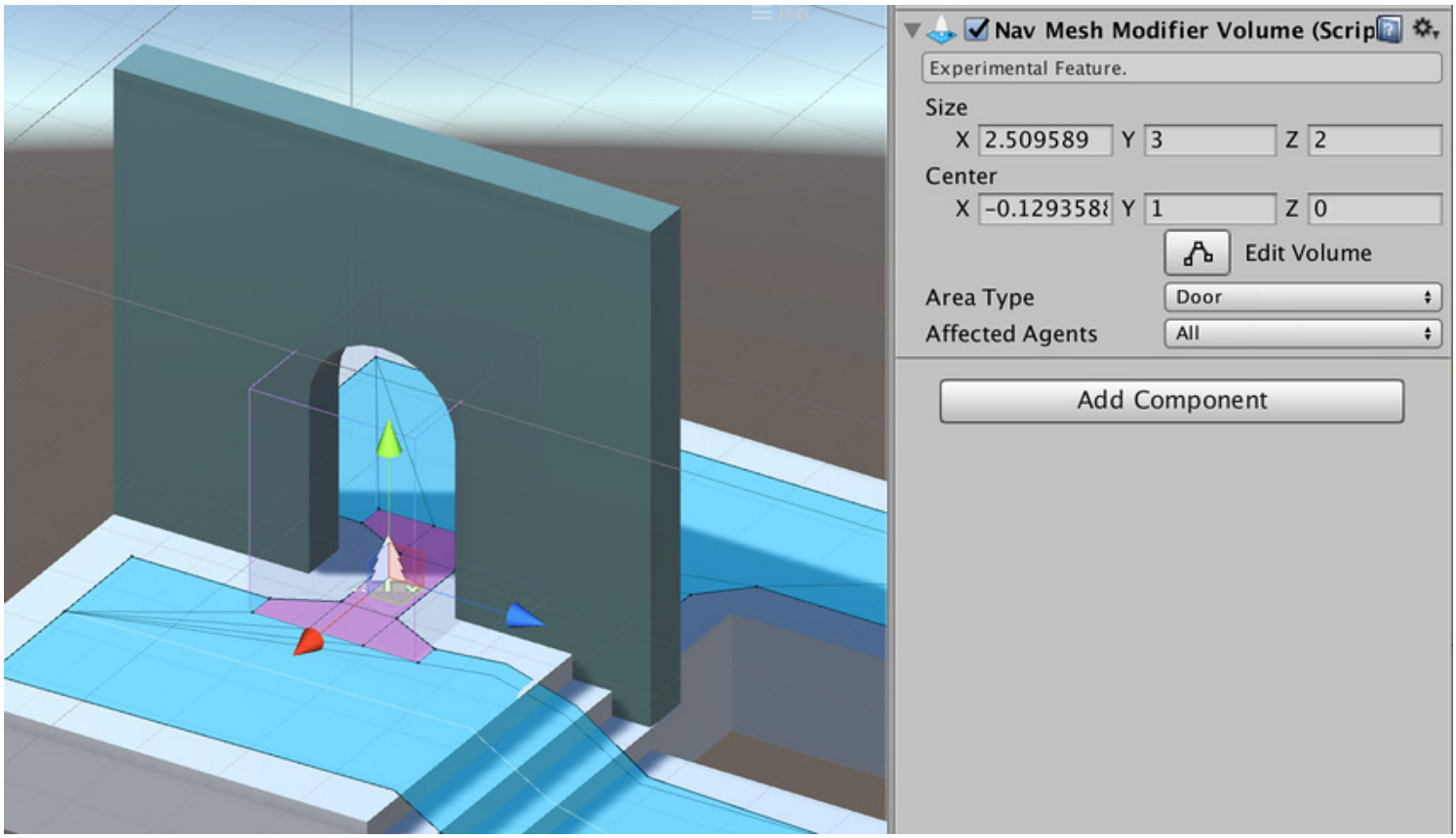
The **NavMesh** Modifier Volume **component** is not in the Unity standard install; see documentation on high-level NavMesh building components for information on how to access it.

NavMesh Modifier Volume marks a defined area as a certain type (for example, **Lava** or **Door**). Whereas NavMesh Modifier marks certain GameObjects with an area type. NavMesh Modifier Volume allows you to change an area type locally based on a specific volume.

To use the NavMesh Modifier Volume component, navigate to **GameObject** > **AI** > **NavMesh Modifier Volume**.

NavMesh Modifier Volume is useful for marking certain areas of walkable surfaces that might not be represented as separate geometry, for example danger areas. You can also use It to make certain areas non-walkable.

The NavMesh Modifier Volume also affects the NavMesh generation process, meaning the NavMesh has to be updated to reflect any changes to NavMesh Modifier Volumes.



A NavMesh Modifier Volume component open in the Inspector window

Property	Function
Size	Dimensions of the NavMesh Modifier Volume, defined by XYZ measurements.
Center	The center of the NavMesh Modifier Volume relative to the GameObject center, defined by XYZ measurements.
Area Type	Describes the area type to which the NavMesh Modifier Volume applies. - **Walkable** (this is the default option) - **Not Walkable** - **Jump**

Property	Function
Affected Agents	A selection of Agents the NavMesh Modifier Volume affects. For example, you may choose to make the selected NavMesh Modifier Volume a danger zone for specific Agent types only. - **None** - **All** (this is the default option) - **Humanoid** - **Ogre**

2017–05–26 Page published with limited editorial review

New feature in 5.6

NavMesh Link

Leave feedback

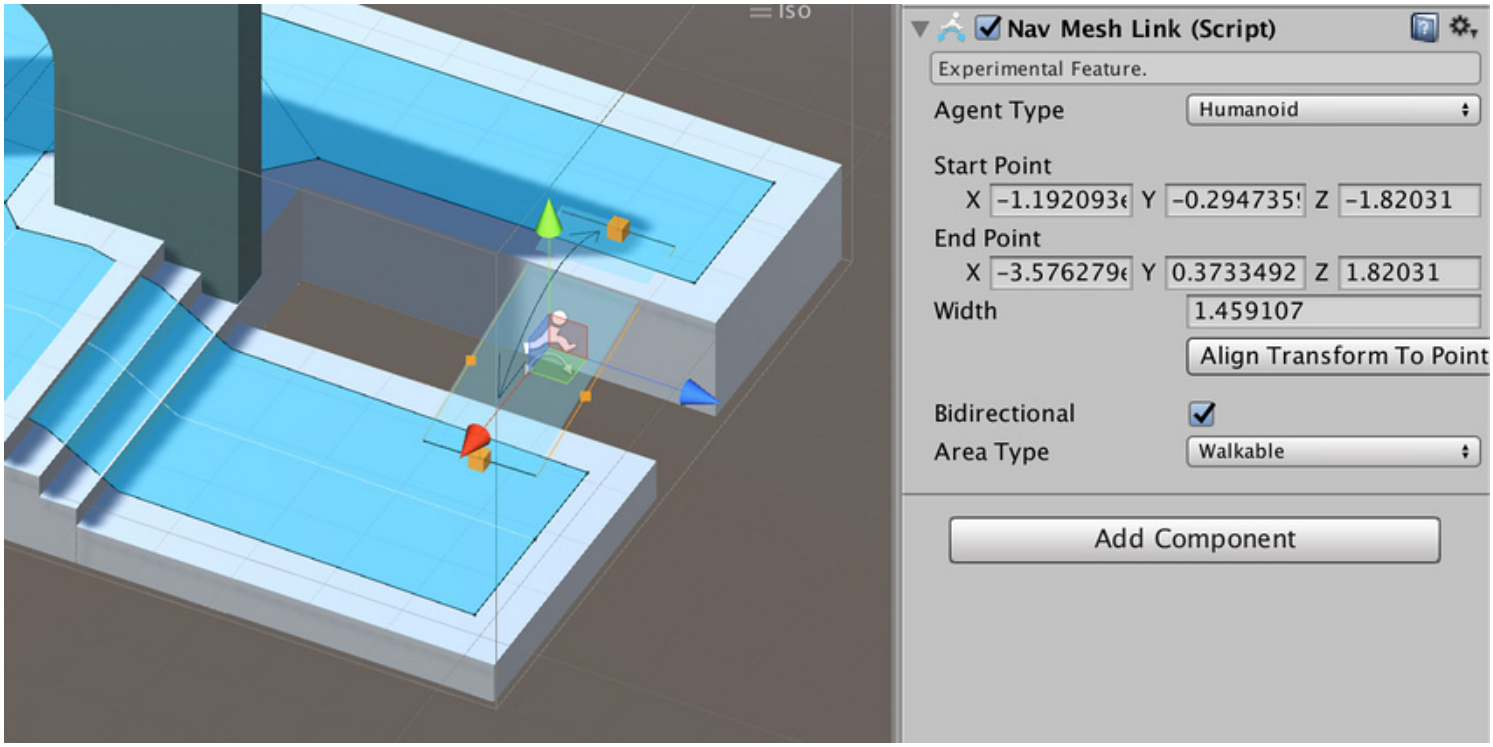
The **NavMesh** Link **component** is not in the Unity standard install; see documentation on high-level NavMesh building components for information on how to access it.

NavMesh Link creates a navigable link between two locations that use NavMeshes.

This link can be from point to point or it can span a gap, in which case the Agent uses the nearest location along the entry edge to cross the link.

You must use a NavMesh Link to connect different NavMesh Surfaces.

To use the NavMesh Link component, navigate to **GameObject** > **AI** > **NavMesh Link**.

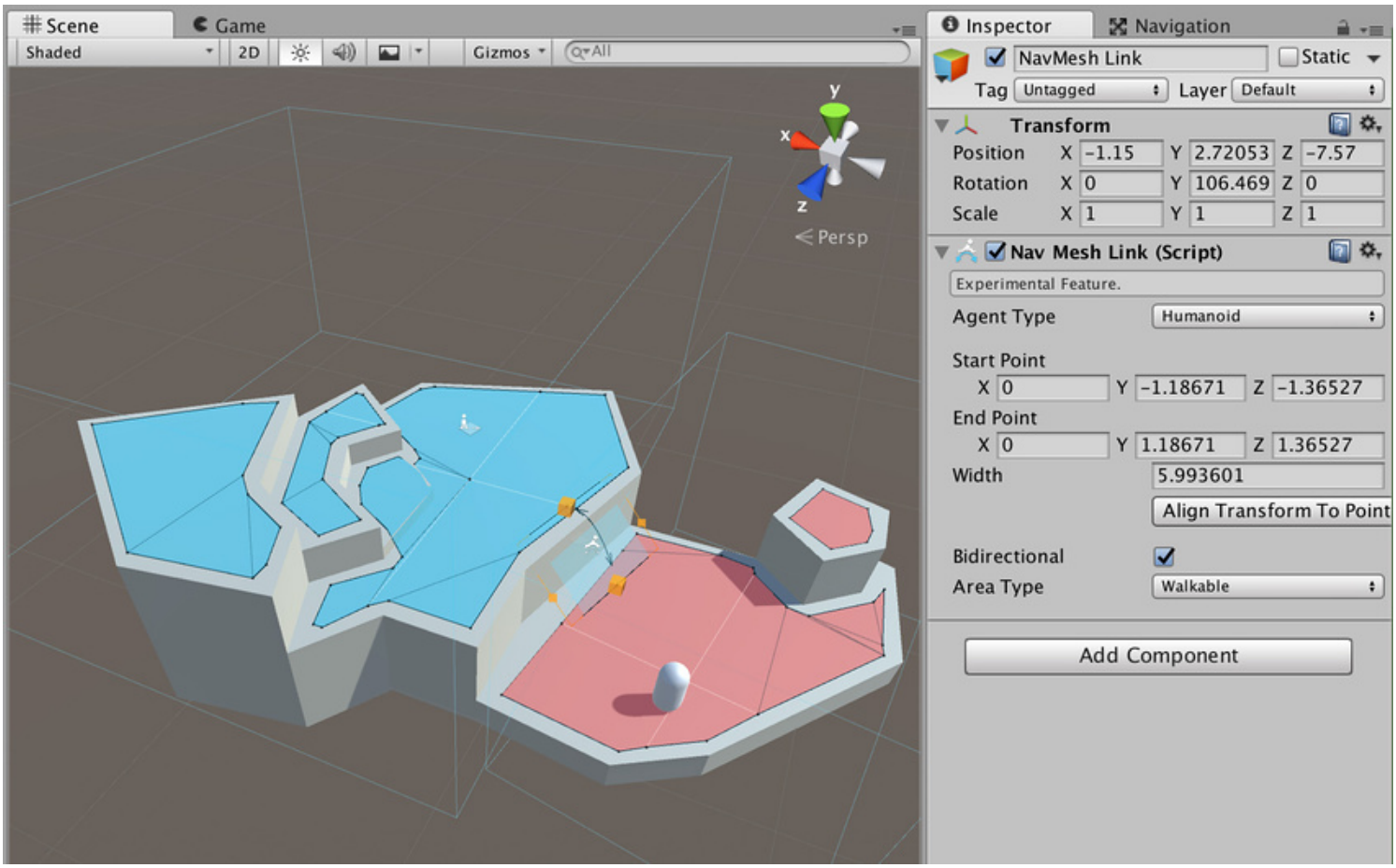


A NavMesh Link component open in the Inspector window

Property	Function
Agent Type	The Agent type that can use the link. - **Humanoid** - **Ogre**
Start Point	The start point of the link, relative to the GameObject. Defined by XYZ measurements.
End Point	The end point of the link, relative to the GameObject. Defined by XYZ measurements.
Align Transform To Points	Clicking this button moves the GameObject at the link's center point and aligns the transform's forward axis with the end point.

Property	Function
Bidirectional	With this tickbox checked, NavMesh Agents traverse the NavMesh Link both ways (from the start point to the end point, and from the end point back to the start point). When this tickbox is unchecked, the NavMesh Link only functions one-way (from the start point to the end point only).
Area Type	The area type of the NavMesh Link (this affects pathfinding costs). - **Walkable** (this is the default option) - **Not Walkable** - **Jump**

Connecting multiple NavMesh Surfaces together



In this image, the blue and red NavMeshes are defined in two different NavMesh Surfaces and connected by a NavMesh Link

If you want an Agent to move between multiple NavMesh Surfaces in a **Scene**, they must be connected using a NavMesh Link.

In the example Scene above, the blue and red NavMeshes are defined in different NavMesh Surfaces, with a NavMesh link connecting them.

Note that when connecting NavMesh Surfaces:

You can connect NavMesh Surfaces using multiple NavMesh Links.

Both the NavMesh Surfaces and the NavMesh Link must have same Agent type.

The NavMesh Link's start and end point must only be on one NavMesh Surface - be careful if you have multiple NavMeshes at the same location.

If you are loading a second NavMesh Surface and you have unconnected NavMesh Links in the first Scene, check that they do not connect to any unwanted NavMesh Surfaces.

2017–05–26 Page published with limited editorial review

New feature in 5.6

NavMesh building components API

Leave feedback

NavMesh building components provide you with additional controls for building (also known as baking) and using NavMeshes at run time and in the Unity Editor.

NavMesh Modifiers are not in the Unity standard install; see documentation on high-level NavMesh building components for information on how to access them.

NavMeshSurface

Properties

`agentTypeID` – ID describing the Agent type the NavMesh should be built for.
`collectObjects` – Defines how input geometry is collected from the **Scene**, one of `UnityEngine.AI.CollectObjects`:
`All` – Use all objects in the scene.
`Volume` – Use all **GameObjects** in the Scene that touch the **bounding volume** (see `size` and `center`)
`Children` – use all objects which are children to the Game Object where the NavMesh Surface is attached to.
`size` – Dimensions of the build volume. The size is not affected by scaling.
`center` – Center of the build volume relative to the transform center.
`layerMask` – Bitmask defining the layers on which the GameObjects must be to be included in the baking.
`useGeometry` – Defines which geometry is used for baking, one of `UnityEngine.NavMeshCollectGeometry`:
`RenderMeshes` – Use geometry from render meshes and terrains
`PhysicsColliders` – Use geometry from **colliders** and **terrains**.
`defaultArea` – Default area type for all input geometries, unless otherwise specified.
`ignoreNavMeshAgent` – True if GameObjects with a Nav **Mesh** Agent component should be ignored as input.
`ignoreNavMeshObstacle` – True if GameObjects with a Nav Mesh Obstacle component should be ignored as input.
`overrideTileSize` – True if tile size is set.
`tileSize` – Tile size in **voxels** (the component description includes information on how to choose tile size).
`overrideVoxelSize` – True if the voxel size is set.
`voxelSize` – Size of the voxel in world units (the component description includes information on how to choose tile size).
`buildHeightMesh` – Not implemented.
`bakedNavMeshData` – Reference to the NavMeshData the surface uses, or null if not set.
`activeSurfaces` – List of all active NavMeshSurfaces.
Note: The above values affect how the bake results, and so you must call `Bake()` to include them.

Public Functions

`void Bake ()`

Bakes a new NavMeshData based on the parameters set on NavMesh Surface. The data can be accessed via `bakedNavMeshData`.

NavMesh Modifier

Properties

`overrideArea` – True if the modifier overrides area type.
`area` – New area type to apply.
`ignoreFromBuild` – True if the GameObject which contains the modifier and its children should be not be used to NavMesh baking.
`activeModifiers` – List of all active NavMeshModifiers.

Public Functions

`bool AffectsAgentType(int agentTypeID)`

Returns true if the modifier applies to the specified Agent type, otherwise false.

NavMesh Modifier Volume

Properties

`size` – Size of the bounding volume in local space units. Transform affects the size.
`center` – Center of the bounding volume in local space units. Transform affects the center.
`area` – Area type to apply for the NavMesh areas that are inside the bounding volume.

Public Functions

`bool AffectsAgentType(int agentTypeID)`

Returns true of the the modifier applies for the specified Agent type.

NavMesh Link

Properties

`agentTypeID` – The type of Agent that can use the link.
`startPoint` – Start point of the link in local space units. Transform affects the location.
`endPoint` – End point of the link in local space units. Transform affects the location.
`width` – Width of the link in world length units.
`bidirectional` – If true, the link can be traversed both ways. If false, the link can be traversed only from start to end.
`autoUpdate` – If true, the link updates the end points to follow the transform of the GameObject every frame.
`area` – Area type of the link (used for pathfinding cost).

Public Functions

`void UpdateLink()`

Updates the link to match the associated transform. This is useful for updating a link, for example after changing the transform position, but is not necessary if the `autoUpdate` property is enabled. However calling `UpdateLink` can have a much smaller performance impact if you rarely change the link transform.

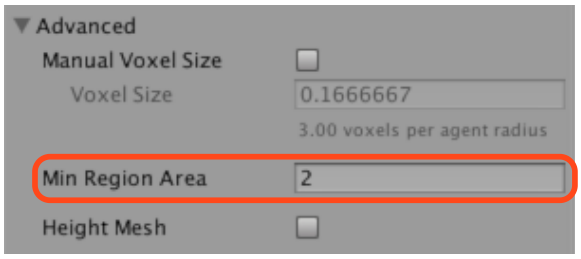
2017–05–26 Page published with limited editorial review

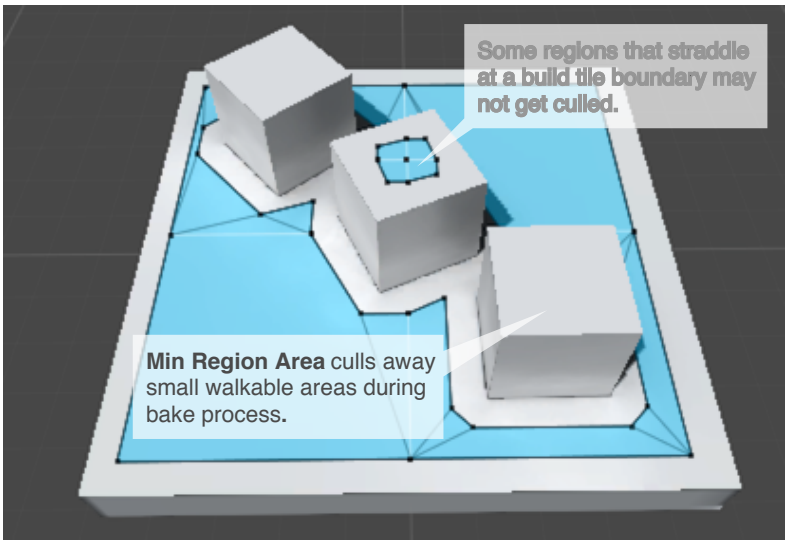
New feature in 5.6

Advanced NavMesh Bake Settings

Leave feedback

Min Region Area





The *Min Region Area* advanced build settings allows you to cull away small non-connected **NavMesh** regions. **NavMesh** regions whose surface area is smaller than the specified value, will be removed.

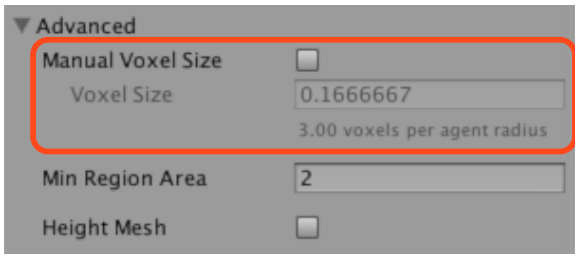
Please note that some areas may not get removed despite the *Min Region Area* setting. The NavMesh is built in parallel as a grid of tiles. If an area straddles a tile boundary, the area is not removed. The reason for this is that the area pruning happens at a stage in the build process where surrounding tiles are not accessible.

Voxel Size

Manual **voxel** size allows you to change the accuracy at which the bake process operates.

The NavMesh bake process uses voxelization to build the NavMesh from arbitrary level geometry. In the first pass of the algorithm, the **scene** is rasterized into voxels, then the walkable surfaces are extracted, and finally the walkable surfaces are turned into a navigation **mesh**. The voxel size describes how accurately the resulting NavMesh represents the **scene** geometry.

The default accuracy is set so that there are 3 voxels per agent radius, that is, the whole agent width is 6 voxels. This is a good trade off between accuracy and bake speed. Halving the voxel size will increase the memory usage by 4x and it will take 4x longer to build the scene.



Generally you should not need to adjust the voxel size, there are two scenarios where this might be necessary: building a *smaller agent radius*, or *more accurate* NavMesh.

Smaller Agent Radius

When you bake for an artificially smaller agent radius, the NavMesh bake system will also reduce the voxel size. If your other agent dimensions stays the same, it may not be necessary to increase the NavMesh build resolution.

The easiest way to do that is as follows:

Set the *Agent Radius* to the real agent radius.
Turn on the *Manual Voxel Size*, this will take the current voxel size and “freeze it”.
Set the artificially smaller *Agent Radius*, since you have checked on the *Manual Voxel Size* the voxel size will not change.

More Accurate NavMesh

If your level has a lot of tight spots, you may want to increase the accuracy by making the voxel smaller. The label under the Voxel Size shows the relation between the voxel size and Agent Radius. A good range is something between 2–8, going further than that generally results really long build times.

When you intentionally build tight corridors in your game, please note that you should leave at least *4 * voxelSize* clearance in addition to the agent radius, especially if the corridors are at angles.

If you need smaller corridors than the NavMesh baking can support, please consider using Off-Mesh Links. These have the additional benefit that you can detect when they are being used and can, for example, play a specific animation.

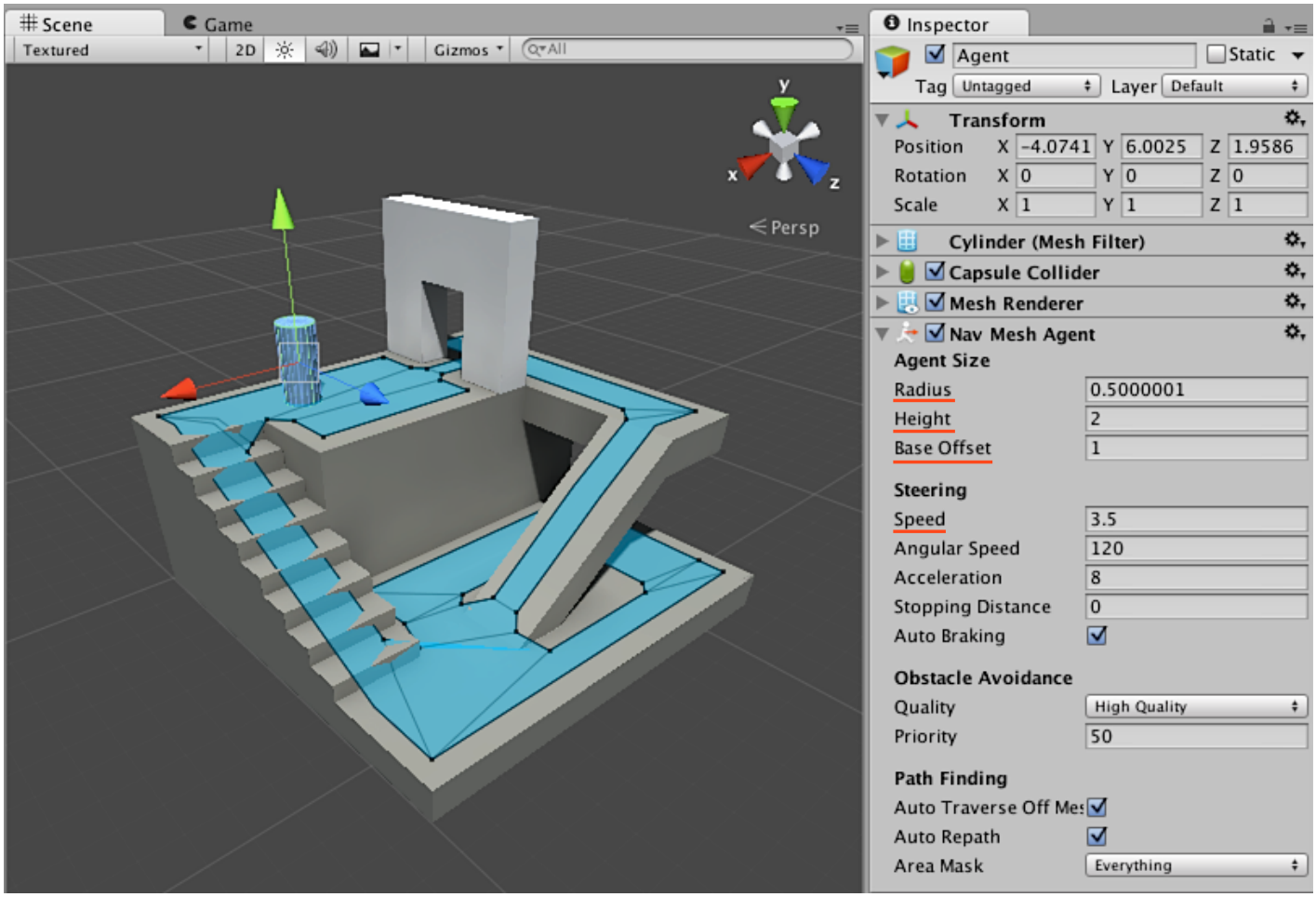
Further reading

Building a NavMesh – workflow for NavMesh baking.
Building Off-Mesh Links Automatically - further details on automatic Off-Mesh Link generation.
Building Height Mesh for Accurate Charater Placement – workflow for Height Mesh baking.

Creating a NavMesh Agent

Leave feedback

Once you have a **NavMesh** baked for your level it is time to create a character which can navigate the **scene**. We're going to build our prototype agent from a cylinder and set it in motion. This is done using a **NavMesh** Agent **component** and a simple script.



First let's create the character:

Create a **cylinder**: **GameObject > 3D Object > Cylinder**.
The default cylinder dimensions (height 2 and radius 0.5) are good for a humanoid shaped agent, so we will leave them as they are.
Add a **NavMesh Agent** component: **Component > Navigation > NavMesh Agent**.
Now you have simple NavMesh Agent set up ready to receive commands!

When you start to experiment with a NavMesh Agent, you most likely are going to adjust its dimensions for your character size and speed.

The **NavMesh Agent** component handles both the pathfinding and the movement control of a character. In your **scripts**, navigation can be as simple as setting the desired destination point - the NavMesh Agent can handle everything from there on.

```
// MoveTo.cs
using UnityEngine;
using System.Collections;

public class MoveTo : MonoBehaviour {

   public Transform goal;
```

```
        void Start () {
            NavMeshAgent agent = GetComponent<NavMeshAgent>();
            agent.destination = goal.position;
        }
    }
```

Next we need to build a simple script which allows you to send your character to the destination specified by another Game Object, and a Sphere which will be the destination to move to:

Create a new **C# script** (`MoveTo.cs`) and replace its contents with the above script.
Assign the MoveTo script to the character you've just created.
Create a **sphere**, this will be the destination the agent will move to.
Move the sphere away from the character to a location that is close to the NavMesh surface.
Select the character, locate the MoveTo script, and assign the Sphere to the **Goal** property.
Press Play; you should see the agent navigating to the location of the sphere.
To sum it up, in your script, you will need to get a reference to the NavMesh Agent component and then to set the agent in motion, you just need to assign a position to its destination property. The Navigation How Tos will give you further examples on how to solve common gameplay scenarios with the NavMesh Agent.

Further Reading

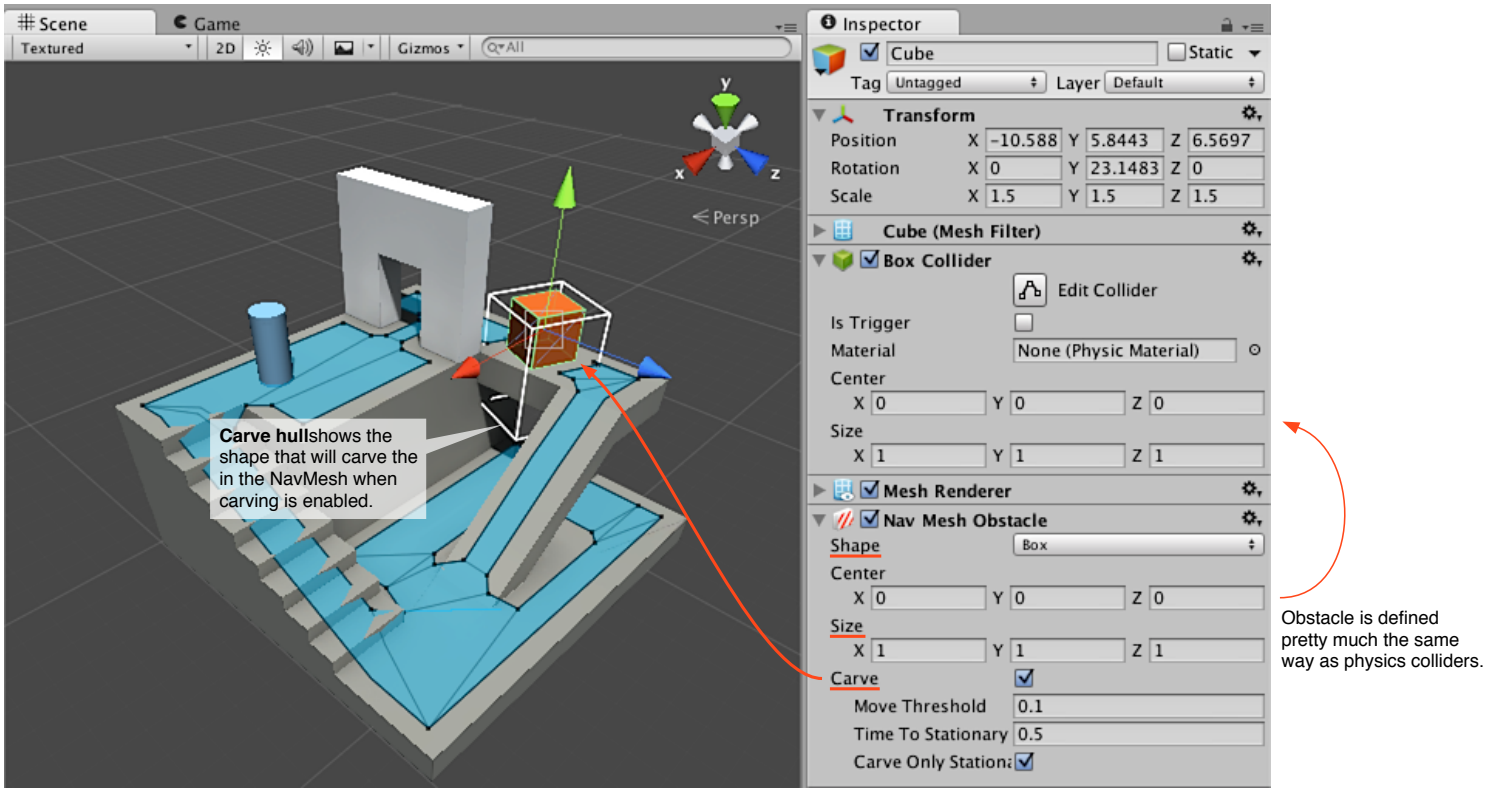
Navigation HowTos - common use cases for NavMesh Agent, with source code.
Inner Workings of the Navigation System - learn more about path following.
NavMesh Agent component reference – full description of all the NavMeshAgent properties.
NavMesh Agent scripting reference - full description of the NavMeshAgent scripting API.

Creating a NavMesh Obstacle

Leave feedback

NavMesh Obstacle **components** can be used to describe obstacles the agents should avoid while navigating. For example the agents should avoid physics controlled objects, such as crates and barrels while moving.

We're going to add a crate to block the pathway at the top of the level.



First create a **cube** to depict the crate: **Game Object > 3D Object > Cube**.
Move the cube to the platform at the top, the default size of the cube is good for a crate so leave it as it is.
Add a **NavMesh Obstacle component** to the cube. Choose **Add Component** from the inspector and choose **Navigation > NavMesh Obstacle**.
Set the shape of the obstacle to **box**, changing the shape will automatically fit the center and size to the render **mesh**.
Add a **Rigid body** to the obstacle. Choose **Add Component** from the inspector and choose **Physics > Rigid Body**.
Finally turn on the **Carve** setting from the **NavMesh** Obstacle **inspector** so that the agent knows to find a path around the obstacle.

Now we have a working crate that is physics controlled, and which the AI knows how to avoid while navigating.

Further Reading

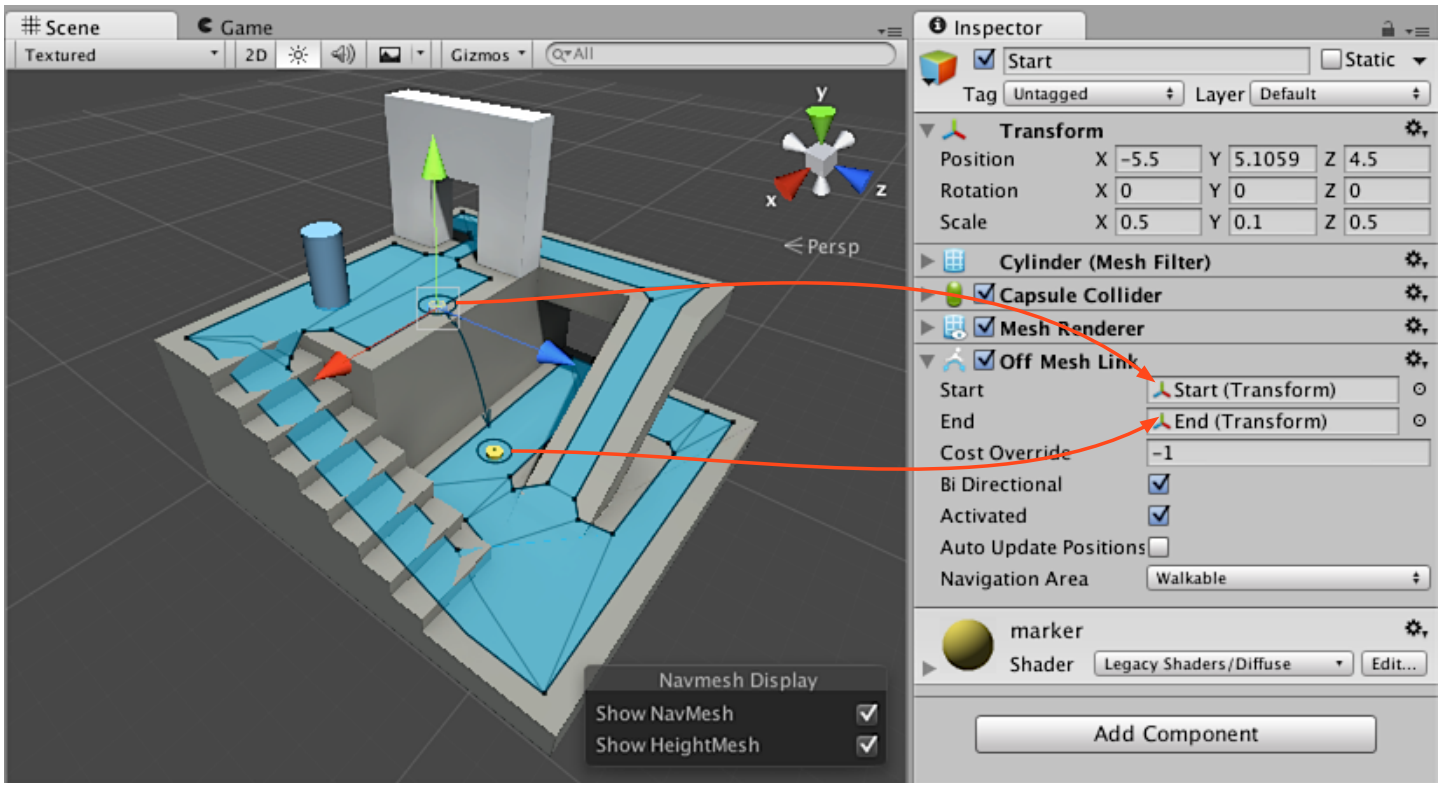
Inner Workings of the Navigation System - learn more about how obstacles are used as part of navigation.
NavMesh Obstacle component reference – full description of all the NavMesh Obstacle properties.
NavMesh Obstacle scripting reference - full description of the NavMesh Obstacle scripting API.

Creating an Off-mesh Link

Leave feedback

Off-Mesh Links are used to create paths crossing outside the walkable navigation **mesh** surface. For example, jumping over a ditch or a fence, or opening a door before walking through it, can be all described as Off-mesh links.

We're going to add an Off-Mesh Link **component** to describe a jump from the upper platform to the ground.



First create **two cylinders**: **Game Object > 3D Object > cylinder**.
You can scale the cylinders to *(0.1, 0.5, 0.1)* to make it easier to work with them.
Move the **first cylinder** at the edge of the top platform, close to the **NavMesh** surface.
Place the **second cylinder** on the ground, close to the NavMesh, at the location where the link should land.
Select the cylinder on the left and add an Off-Mesh Link component to it. Choose **Add Component** from the inspector and choose **Navigation > Off Mesh Link**.
Assign the leftmost cylinder in the **Start** field and rightmost cylinder in the **End** field.
Now you have functioning Off-Mesh Link set up! If the path via the off-mesh link is shorter than via walking along the Navmesh, the off-mesh link will be used.

You can use any game object in the **scene** to hold the Off-Mesh link component, for example a fence **prefab** could contain the off-mesh link component. Similarly you can use any game object with a Transform as the start and end marker.

The NavMesh bake process can detect and create common jump-across and drop-down links automatically. Take a look at the Building Off-Mesh Links Automatically for more details.

Further Reading

Building Off-Mesh Links Automatically - learn how to automatically build Off-Mesh Links.
Navigation HowTos - common use cases for NavMesh Agent, with source code.
Off-Mesh Link component reference – full description of all the Off-Mesh Link properties.
Off-Mesh Link scripting reference - full description of the Off-Mesh Link scripting API.

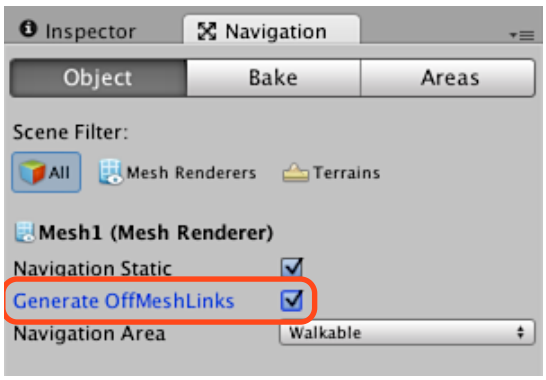
Building Off-Mesh Links Automatically

Leave feedback

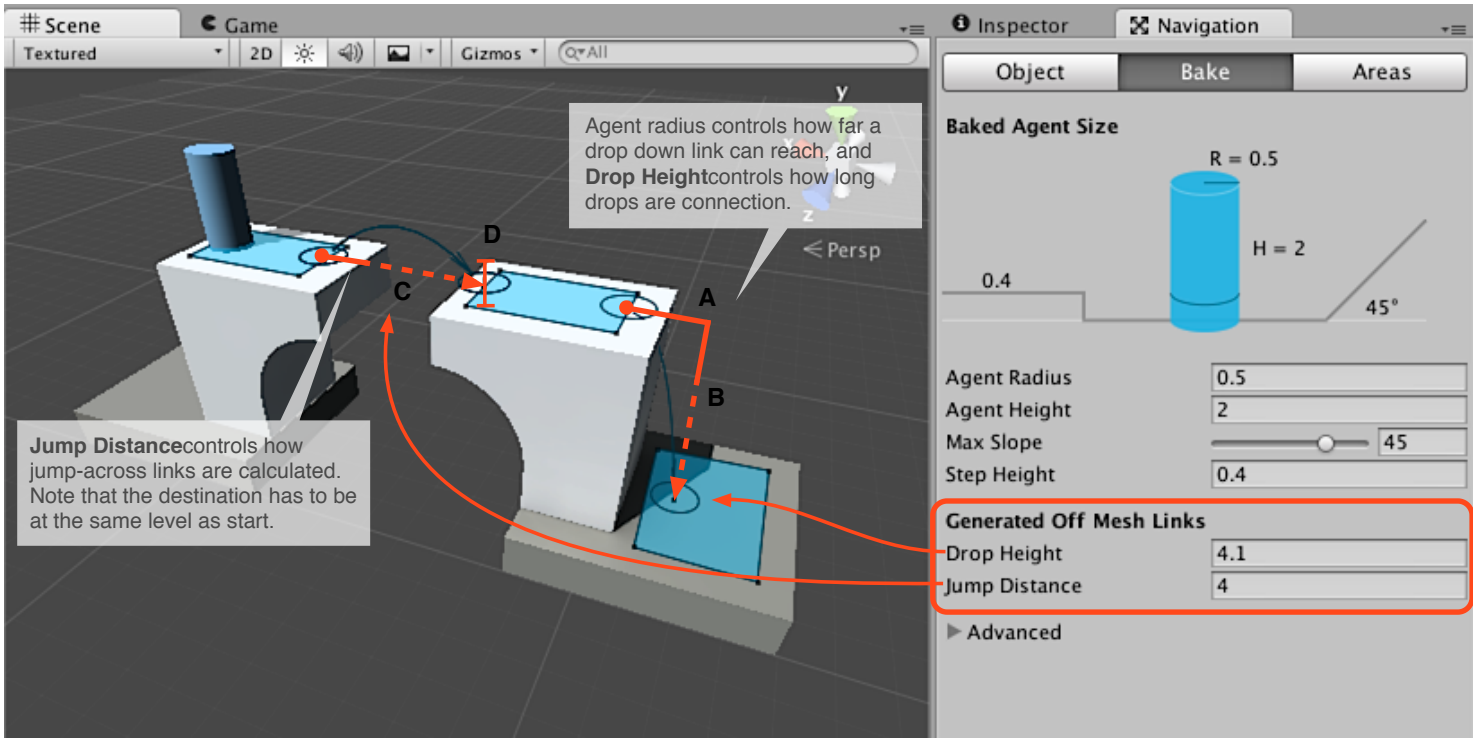
Some use cases for Off-Mesh Links can be detected automatically. The two most common ones are: *Drop-Down* and *Jump-Across*.

Drop-Down links are created to drop down from a platform.
Jump-Across links are created to jump across a crevice.
In order to find the jump locations automatically, the build process walks along the edges of the **NavMesh** and checks if the landing location of the jump is on **NavMesh**. If the jump trajectory is unobstructed an Off-Mesh link is created.

Let's set up automatic Off-Mesh Link generation. If you're not familiar with NavMesh baking, take a look at Building a NavMesh.



First, the objects in the **Scene** where the jump can *start from* needs to be marked. This is done my checking **Generate Off-Mesh Links** option in the *Navigation Window* under *Objects* tab.



The second step is to setup the drop-down and jump-across trajectories:

Drop-Down link generation is controlled by the Drop Height parameter. The parameter controls what is the highest drop that will be connected, setting the value to 0 will disable the generation.
The trajectory of the drop-down link is defined so that the horizontal travel **(A)** is: *2*agentRadius + 4*voxelSize*. That is, the drop will land just beyond the edge of the platform. In addition the vertical travel **(B)** needs to be more than bake settings' *Step Height* (otherwise we could just step down) and less than *Drop Height*. The adjustment by **voxel** size is done so that any round off errors during voxelization does not prevent the links

being generated. You should set the *Drop Height* to a bit larger value than what you measure in your level, so that the links will connect properly.
Jump-Across link generation is controlled by the *Jump Distance* parameter. The parameter controls what is the furthest distance that will be connected. Setting the value to 0 will disable the generation.
The trajectory of the jump-across link is defined so that the horizontal travel **(C)** is more than *2*agentRadius* and less than Jump Distance. In addition the landing location **(D)** must not be further than voxelSize from the level of the start location.

Now that objects are marked, and settings adjusted, it's time to press *Bake* and you have will have automatically generated off-mesh links! When ever you change the scene and bake, the old links will be discarded and new links will be created based on the new scene.

Trouble Shooting

Things to keep in mind if Off-Mesh links are not generated at locations where you expect them to be:

Drop Height should a bit bigger than the actual distance measured in your level. This ensures that small deviations that happen during the NavMesh baking process will not prevent the link to be connected.
Jump Distance should be a bit longer than the actual distance measured in your level. The Jump Distance is measured from one location on a NavMesh to another location on the NavMesh, which means that you should add *2*agentRadius* (plus a little) to make sure the crevices are crossed.

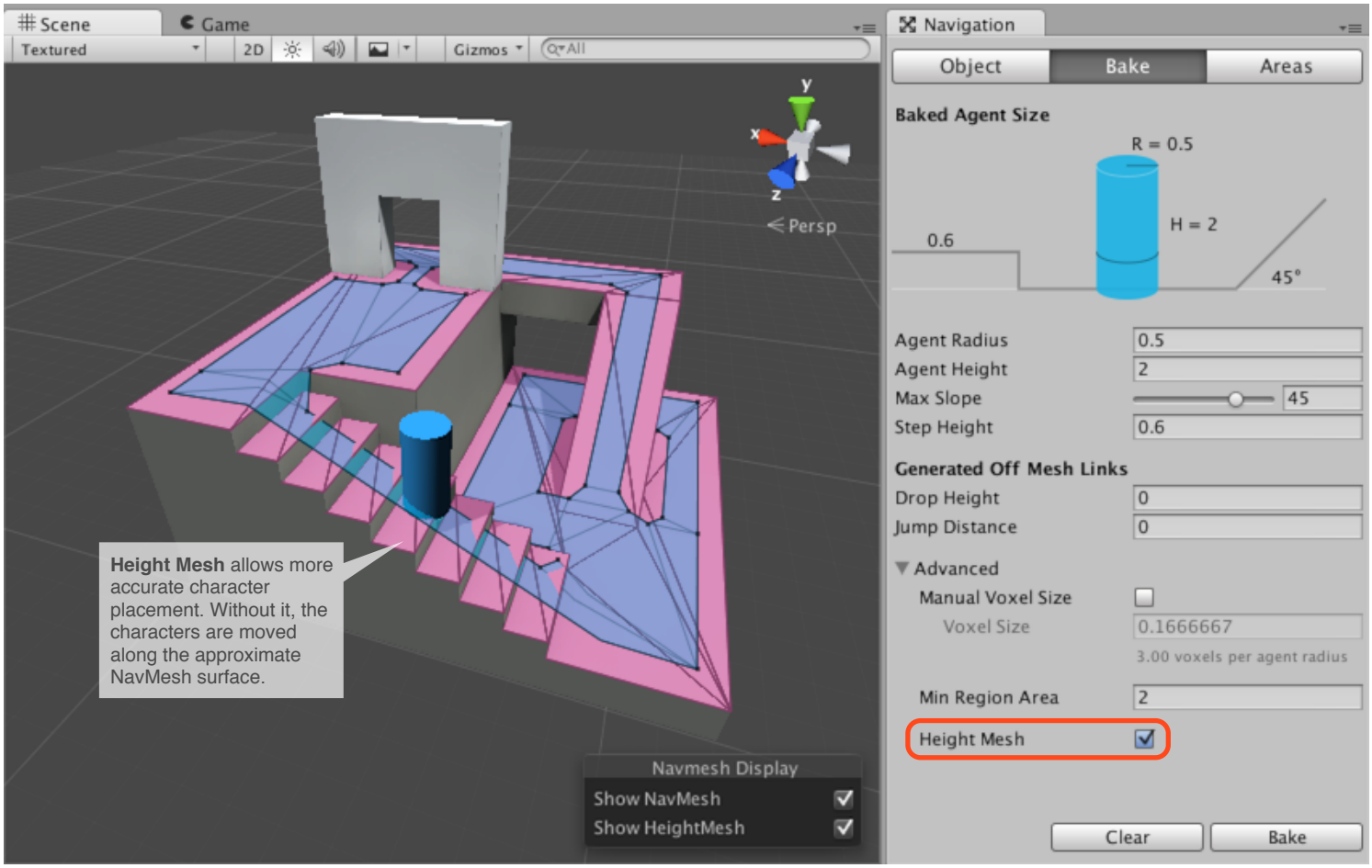
Further Reading

Creating Off-Mesh Links - learn how to manually create Off-Mesh Links.
Building a NavMesh – workflow for NavMesh baking.
Bake Settings – full description of the NavMesh bake settings.
Off-Mesh Link component reference – full description of all the Off-Mesh Link properties.
Off-Mesh Link scripting reference - full description of the Off-Mesh Link scripting API.

Building Height Mesh for Accurate Character Placement

Leave feedback

Height **mesh** allows you to place your character more accurately on the walkable surfaces.



While navigating, the **NavMesh** Agent is constrained on the surface of the **NavMesh**. Since the **NavMesh** is an approximation of the walkable space, some features are evened out when the **NavMesh** is being built. For example, stairs may appear as a slope in the **NavMesh**. If your game requires accurate placement of the agent, you should enable *Height Mesh* building when you bake the **NavMesh**. The setting can be found under the Advanced settings in Navigation window. Note that building Height Mesh will take up memory and processing at runtime, and it will take a little longer to bake the **NavMesh**.

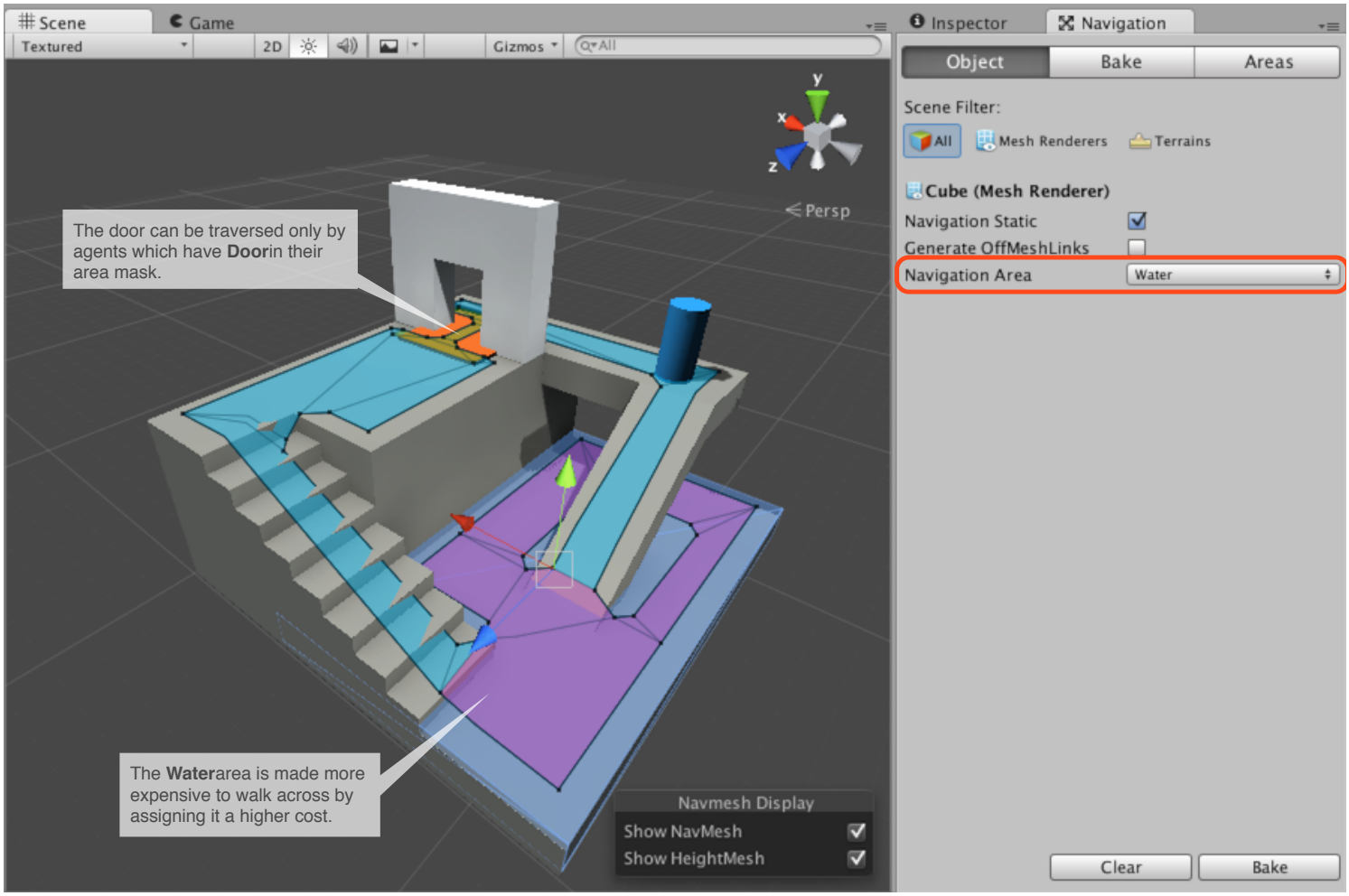
Further reading

Building a NavMesh – workflow for NavMesh baking.

Navigation Areas and Costs

Leave feedback

The *Navigation Areas* define how difficult it is to walk across a specific area, the lower cost areas will be preferred during path finding. In addition each **NavMesh** Agent has an *Area Mask* which can be used to specify on which areas the agent can move.



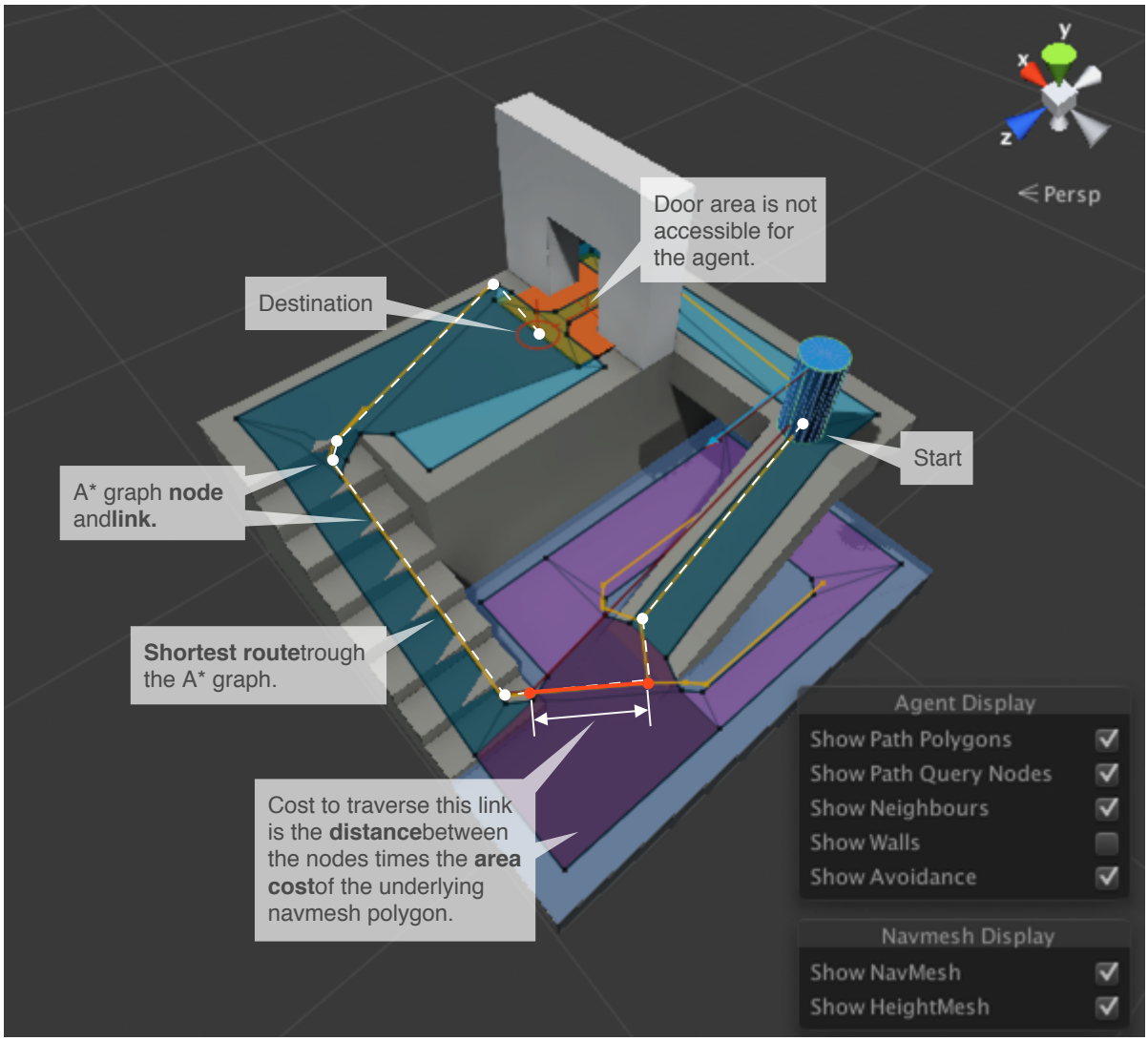
In the above example the area types are used for two common use cases:

Water area is made more costly to walk by assigning it a higher cost, to deal with a scenario where walking on shallow water is slower.
Door area is made accessible by specific characters, to create a scenario where humans can walk through doors, but zombies cannot.
The area type can be assigned to every object that is included in the NavMesh baking, in addition, each Off-Mesh Link has a property to specify the area type.

Pathfinding Cost

In a nutshell, the cost allows you to control which areas the pathfinder favors when finding a path. For example, if you set the cost of an area to 3.0, traveling across that area is considered to be three times longer than alternative routes.

To fully understand how the cost works, let's take a look at how the pathfinder works.



Nodes and links visited during pathfinding.

Unity uses A* to calculate the shortest path on the NavMesh. A* works on a graph of connected nodes. The algorithm starts from the nearest node to the path start and visits the connect nodes until the destination is reached.

Since the Unity navigation representation is a **mesh** of polygons, the first thing the pathfinder needs to do is to place a point on each polygon, which is the location of the node. The shortest path is then calculated between these nodes.

The yellow dots and lines in the above picture shows how the nodes and links are placed on the NavMesh, and in which order they are traversed during the A*.

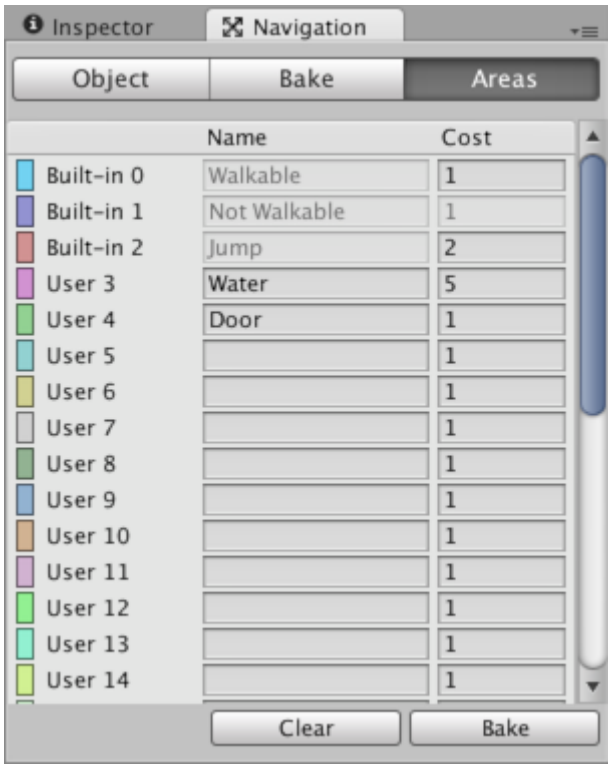
The cost to move between two nodes depends on the distance to travel and the cost associated with the area type of the polygon under the link, that is, *distance * cost*. In practice this means, that if the cost of an area is 2.0, the distance across such polygon will appear to be twice as long. The A* algorithm requires that all costs must be larger than 1.0.

The effect of the costs on the resulting path can be hard to tune, especially for longer paths. The best way to approach costs is to treat them as hints. For example, if you want the agents to not to use Off-Mesh Links too often, you could increase their cost. But it can be challenging to tune a behavior where the agents to prefer to walk on sidewalks.

Another thing you may notice on some levels is that the pathfinder does not always choose the very shortest path. The reason for this is the node placement. The effect can be noticeable in scenarios where big open areas are next to tiny obstacles, which results navigation mesh with very big and small polygons. In such cases the nodes on the big polygons may get placed anywhere in the big polygon and from the pathfinder's point of view it looks like a detour.

The *cost* per *area type* can be set globally in the *Areas* tab, or you can override them per agent using a script.

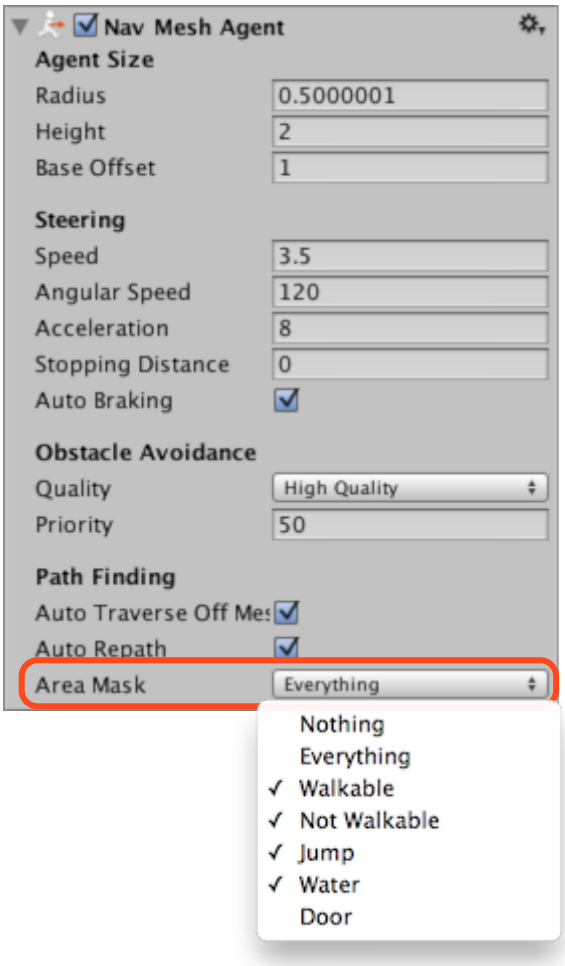
Area Types



The area types are specified in the *Navigation Window*'s *Areas* tab. There are 29 custom types, and 3 built-in types: *Walkable*, *Not Walkable*, and *Jump*.

Walkable is a generic area type which specifies that the area can be walked on.
Not Walkable is a generic area type which prevents navigation. It is useful for cases where you want to mark certain object to be an obstacle, but without getting NavMesh on top of it.
Jump is an area type that is assigned to all auto-generated Off-Mesh Links.
If several objects of different area types are overlapping, the resulting navmesh area type will generally be the one with the highest index. There is one exception however: *Not Walkable* always takes precedence. Which can be helpful if you need to block out an area.

Area Mask



Each agent has an *Area Mask* which describes which areas it can use when navigating. The area mask can be set in the agent properties, or the bitmask can be manipulated using a script at runtime.

The area mask is useful when you want only certain types characters to be able to walk through an area. For example, in a zombie evasion game, you could mark the area under each door with a *Door* area type, and uncheck the Door area from the zombie character's Area Mask.

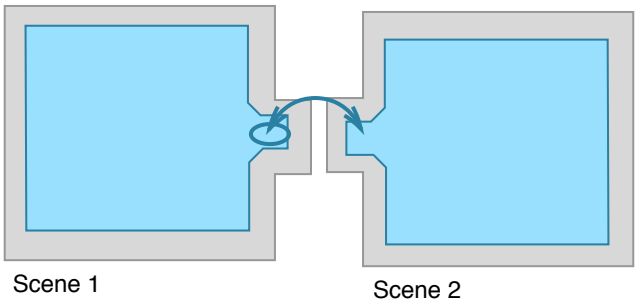
Further Reading

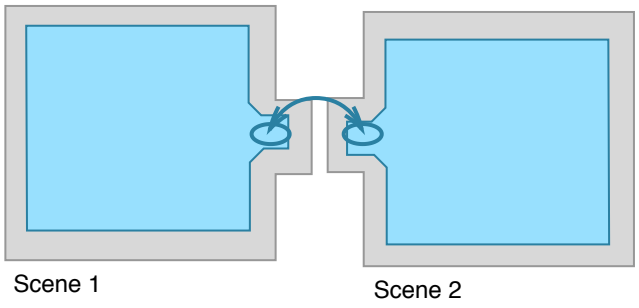
Building a NavMesh – workflow for NavMesh baking.
NavMeshAgent.areaMask - Script API to set areaMask for an agent.
NavMeshAgent.SetAreaCost() - Script API to set area cost for an agent.

Loading Multiple NavMeshes using Additive Loading

Leave feedback

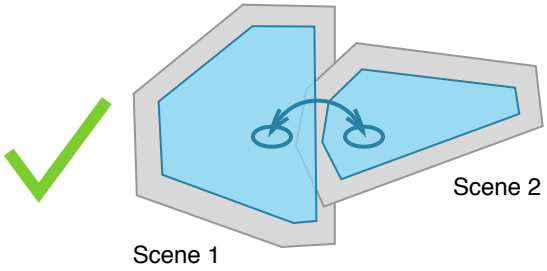
The NavMeshes in different **Scenes** are not connect by default. When you load another level using *Application.LoadLevelAdditive()* you will need to connect the NavMeshes in different Scenes using an Off-Mesh link.

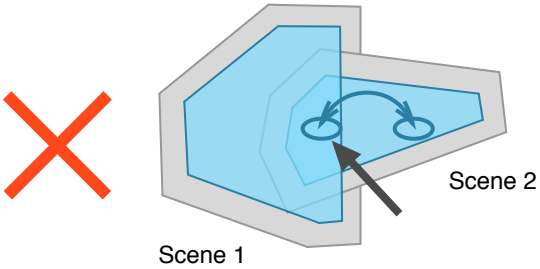




In this example we have *Scene 1* and *Scene 2*. The *Scene 1* has an Off-Mesh Link starting over a walkable area and landing over a walkable area in *Scene 2*. There can be as many Off-Mesh Links connecting the Scenes as necessary.

When authored the other end point of the scene connecting Off-Mesh links are not connected. After a new Scene is loaded, the Off-Mesh Links will be reconnected.





If the multiple scenes have **NavMesh** overlapping at the same area, the position picking may be arbitrary **NavMesh** at that location. This applies to agents, Off-Mesh Links and position picking using the **NavMesh** API. You should create the scene crossing Off-Mesh links so that they start and end clearly on over one **NavMesh** only. Overlapping **NavMesh** areas are not automatically connected.

Further Reading

Building a NavMesh – workflow for NavMesh baking.
Creating Off-Mesh Links - learn how to manually create Off-Mesh Links.

Using NavMesh Agent with Other Components

Leave feedback

You can use **NavMesh** Agent, **NavMesh** Obstacle, and Off **Mesh** Link **components** with other Unity **components** too. Here's a list of dos and don'ts when mixing different **components** together.

NavMesh Agent and Physics

You don't need to add physics **colliders** to NavMesh Agents for them to avoid each other
That is, the navigation system simulates agents and their reaction to obstacles and the static world.
Here the static world is the baked NavMesh.
If you want a NavMesh Agent to push around physics objects or use physics triggers:
Add a Collider component (if not present)
Add **Rigidbody** component
Turn on kinematic (Is Kinematic) - this is important!
Kinematic means that the rigid body is controlled by something else than the physics simulation
If both NavMesh Agent and Rigidbody (non-kinematic) are active at the same time, you have race condition
Both components may try to move the agent at the same which leads to undefined behavior
You can use a NavMesh Agent to move e.g. a player character, without physics
Set players agent's avoidance priority to a small number (high priority), to allow the player to brush through crowds
Move the player agent using NavMeshAgent.velocity, so that other agents can predict the player movement to avoid the player.

NavMesh Agent and Animator

NavMesh Agent and Animator with **Root Motion** can cause race condition
Both components try to move the transform each frame
Two possible solutions
Information should always flow in one direction
Either agent moves the character and animations follows
Or animation moves the character based on simulated result
Otherwise you'll end up having a hard to debug feedback loop
Animation follows agent
Use the NavMeshAgent.velocity as input to the Animator to roughly match the agent's movement to the animations
Robust and simple to implement, will result in foot sliding where animations cannot match the velocity
Agent follows animation
Disable NavMeshAgent.updatePosition and NavMeshAgent.updateRotation to detach the simulation from the game objects locations
Use the difference between the simulated agent's position (NavMeshAgent.nextPosition) and animation root (Animator.rootPosition) to calculate controls for the animations
See Coupling Animation and Navigation for more details

NavMesh Agent and NavMesh Obstacle

Do not mix well!
Enabling both will make the agent trying to avoid itself
If carving is enabled in addition, the agent tries to constantly remap to the edge of the carved hole, even more erroneous behavior ensues
Make sure only one of them are active at any given time
Deceased state, you may turn off the agent and turn on the obstacle to force other agents to avoid it
Alternatively you can use priorities to make certain agents to be avoided more

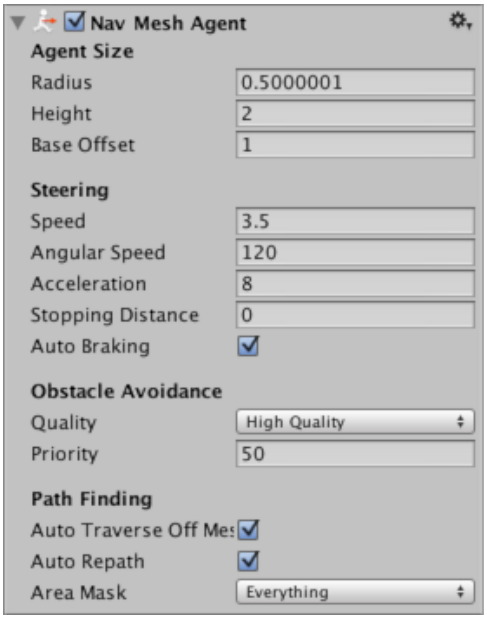
NavMesh Obstacle and Physics

If you want physics controlled object to affect NavMesh Agent's behavior
Add NavMesh Obstacle component to the object which agent should be aware of, this allows the avoidance system to reason about the obstacle
If a game object has a Rigidbody and a NavMesh Obstacle attached, the obstacle's velocity is obtained from the Rigidbody automatically
This allows NavMesh Agents to predict and avoid the moving obstacle

NavMesh Agent

Leave feedback

NavMeshAgent **components** help you to create characters which avoid each other while moving towards their goal. Agents reason about the game world using the **NavMesh** and they know how to avoid each other as well as other moving obstacles. Pathfinding and spatial reasoning are handled using the scripting API of the **NavMesh** Agent.



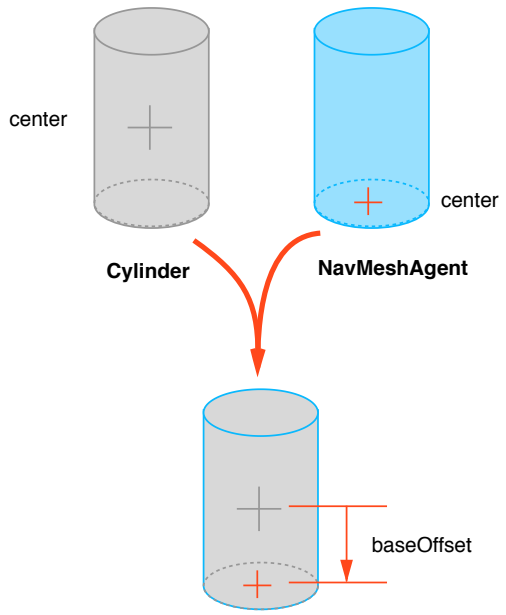
Properties

Property	Function
Agent Size	
Radius	Radius of the agent, used to calculate **collisions** between obstacles and other agents.
Height	The height clearance the agent needs to pass below an obstacle overhead.
Base offset	Offset of the collision cylinder in relation to the transform pivot point.
Steering	
Speed	Maximum movement speed (in world units per second).
Angular Speed	Maximum speed of rotation (degrees per second).
Acceleration	Maximum acceleration (in world units per second squared).
Stopping distance	The agent will stop when this close to the goal location.
Auto Braking	When enabled, the agent will slow down when reaching the destination. You should disable this for behaviors such patrolling, where the agent should move smoothly between multiple points
Obstacle Avoidance	
Quality	Obstacle avoidance quality. If you have high number of agents you can save CPU time by reducing the obstacle avoidance quality. Setting avoidance to none, will only resolve collision, but will not try to actively avoid other agents and obstacles.
Priority	Agents of lower priority will be ignored by this agent when performing avoidance. The value should be in the range 0–99 where lower numbers indicate higher priority.
Path Finding	
Auto Traverse OffMesh Link	Set to true to automatically traverse off-mesh links. You should turn this off when you you want to use animation or some specific way to traverse off-mesh links.

Property	Function
Auto Repath	When enabled the agent will try to find path again when it reaches the end of a partial path. When there is no path to the destination, a partial path is generated to the closest reachable location to the destination.
Area Mask	Area mask describes which area types the agent will consider when finding a path. When you prepare meshes for NavMesh baking, you can set each meshes area type. For example you can mark stairs with special area type, and forbid some character types from using the stairs.

Details

The agent is defined by an upright cylinder whose size is specified by the *Radius* and *Height* properties. The cylinder moves with the object but always remains upright even if the object itself rotates. The shape of the cylinder is used to detect and respond to collisions between other agents and obstacles. When the **GameObject**'s anchor point is not at the base of the cylinder, you can use the *Base Offset* property to take up the height difference.



The height and radius of the cylinder are actually specified in *two* different places: the NavMesh bake settings and the properties of the individual agents.

NavMesh bake settings describe how all the NavMesh Agents are colliding or avoiding the static world geometry. In order to keep memory on budget and CPU load in check, only one size can be specified in the bake settings.
NavMesh Agent properties values describe how the agent collides with moving obstacles and other agents.
Most often you set the size of the agent the same in both places. But, for example, a heavy soldier may have larger radius, so that other agents will leave more space around him, but otherwise he'll avoid the environment just the same.

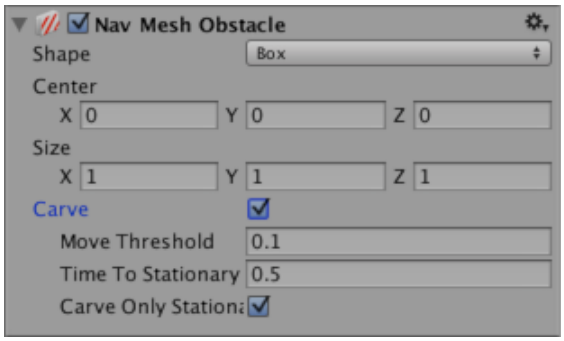
Further Reading

Creating a NavMesh Agent – workflow on how to create a NavMesh Agent.
Inner Workings of the Navigation System - learn more about how obstacles are used as part of navigation.
NavMesh Agent scripting reference - full description of the NavMeshAgent scripting API.

Nav Mesh Obstacle

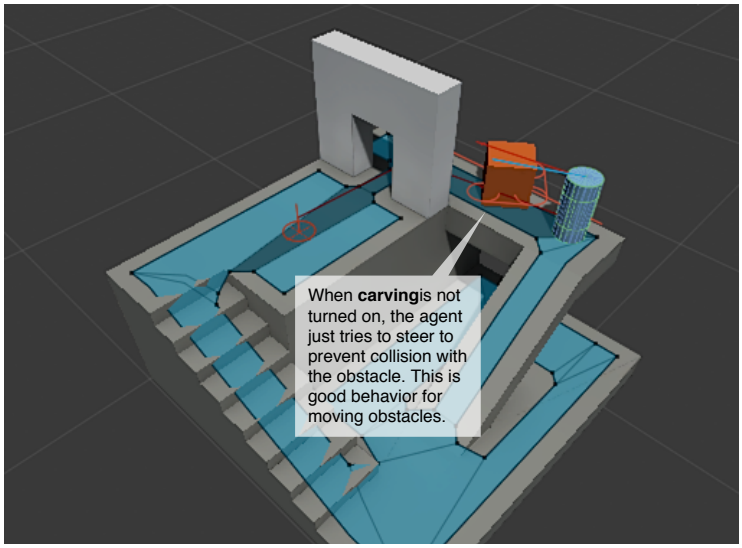
Leave feedback

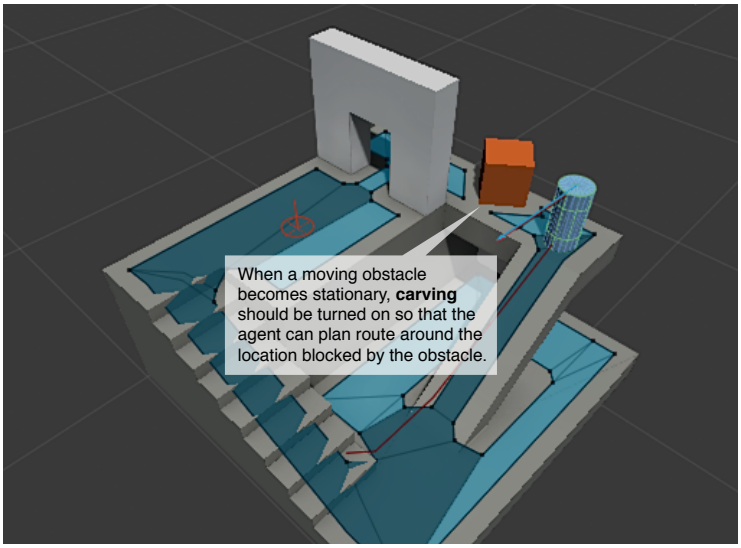
The **Nav** Mesh__ Obstacle__ **component** allows you to describe moving obstacles that Nav Mesh Agents should avoid while navigating the world (for example, barrels or crates controlled by the physics system). While the obstacle is moving, the Nav Mesh Agents do their best to avoid it. When the obstacle is stationary, it carves a hole in the **NavMesh**. Nav Mesh Agents then change their paths to steer around it, or find a different route if the obstacle causes the pathway to be completely blocked.



Property	Function
Shape	The shape of the obstacle geometry. Choose whichever one best fits the shape of the object.
Box	
Center	Center of the box relative to the transform position.
Size	Size of the box.
Capsule	
Center	Center of the capsule relative to the transform position.
Radius	Radius of the capsule.
Height	Height of the capsule.
Carve	When the Carve checkbox is ticked, the Nav Mesh Obstacle creates a hole in the NavMesh.
Move Threshold	Unity treats the Nav Mesh Obstacle as moving when it has moved more than the distance set by the Move Threshold. Use this property to set the threshold distance for updating a moving carved hole.
Time To Stationary	The time (in seconds) to wait until the obstacle is treated as stationary.
Carve Only Stationary	When enabled, the obstacle is carved only when it is stationary. See Logic for moving Nav Mesh Obstacles, below, to learn more.

Details





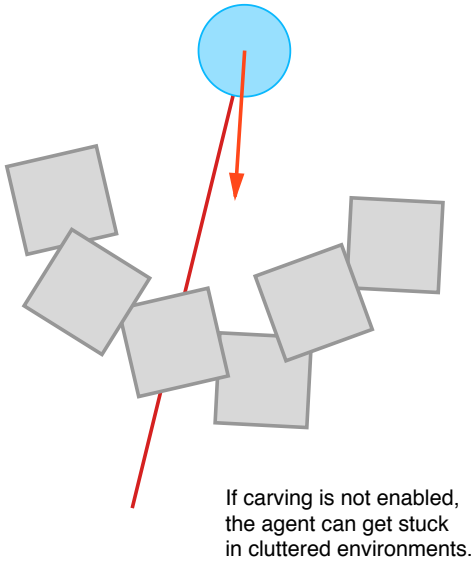
Nav Mesh Obstacles can affect the Nav Mesh Agent's navigation during the game in two ways:

Obstructing

When **Carve** is not enabled, the default behavior of the Nav Mesh Obstacle is similar to that of a **Collider**. Nav Mesh Agents try to avoid **collisions** with the Nav Mesh Obstacle, and when close, they collide with the Nav Mesh Obstacle. Obstacle avoidance behaviour is very basic, and has a short radius. As such, the Nav Mesh Agent might not be able to find its way around in an environment cluttered with Nav Mesh Obstacles. This mode is best used in cases where the obstacle is constantly moving (for example, a vehicle or player character).

Carving

When **Carve** is enabled, the obstacle carves a hole in the NavMesh when stationary. When moving, the obstacle is an obstruction. When a hole is carved into the NavMesh, the pathfinder is able to navigate the Nav Mesh Agent around locations cluttered with obstacles, or find another route if the current path gets blocked by an obstacle. It's good practice to turn on carving for Nav Mesh Obstacles that generally block navigation but can be moved by the player or other game events like explosions (for example, crates or barrels).



Logic for moving Nav Mesh Obstacles

Unity treats the Nav Mesh Obstacle as moving when it has moved more than the distance set by the **Carve** > **Move Threshold**. When the Nav Mesh Obstacle moves, the carved hole also moves. However, to reduce CPU overhead, the hole is only recalculated when necessary. The result of this calculation is available in the next frame update. The recalculation logic has two options:

Only carve when the Nav Mesh Obstacle is stationary

Carve when the Nav Mesh Obstacle has moved

Only carve when the Nav Mesh Obstacle is stationary

This is the default behavior. To enable it, tick the Nav Mesh Obstacle component's **Carve Only Stationary** checkbox. In this mode, when the Nav Mesh Obstacle moves, the carved hole is removed. When the Nav Mesh Obstacle has stopped moving and has been stationary for more than the time set by **Carving Time To Stationary**, it is treated as stationary and the carved hole is updated again. While the Nav Mesh Obstacle is moving, the Nav Mesh Agents avoid it using collision avoidance, but don't plan paths around it.

Carve Only Stationery is generally the best choice in terms of performance, and is a good match when the **GameObject** associated with the Nav Mesh Obstacle is controlled by physics.

Carve when the Nav Mesh Obstacle has moved

To enable this mode, untick the Nav Mesh Obstacle component's **Carve Only Stationary** checkbox. When this is unticked, the carved hole is updated when the obstacle has moved more than the distance set by **Carving Move Threshold**. This mode is useful for large, slowly moving obstacles (for example, a tank that is being avoided by infantry).

Note: When using NavMesh query methods, you should take into account that there is a one-frame delay between changing a Nav Mesh Obstacle and the effect that change has on the NavMesh.

See also

Creating a Nav Mesh Obstacle - Guidance on creating Nav Mesh Obstacles.

Inner Workings of the Navigation System - Learn more about how Nav Mesh Obstacles are used as part of navigation.

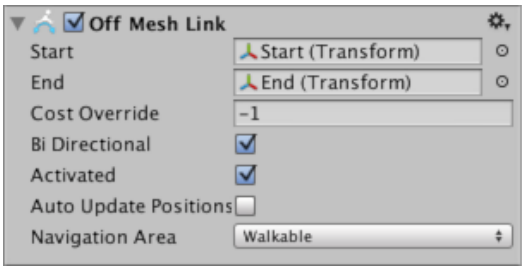
Nav Mesh Obstacle scripting reference - Full description of the Nav Mesh Obstacle scripting API.

Off-Mesh Link

Leave feedback

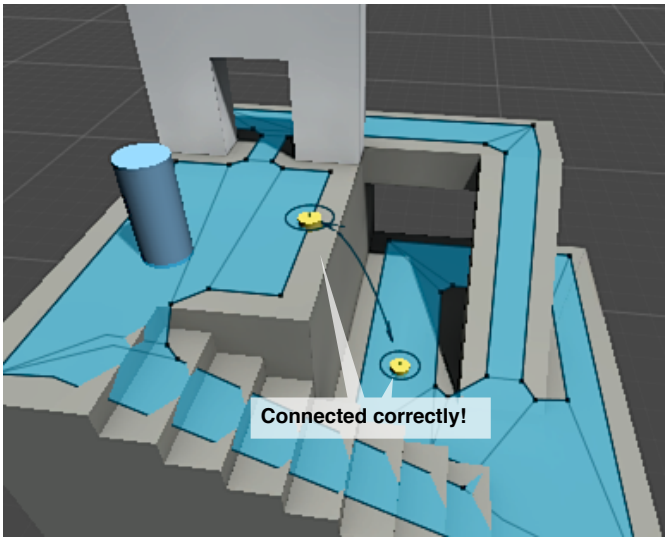
OffMeshLink **component** allows you to incorporate navigation shortcuts which cannot be represented using a walkable surface. For example, jumping over a ditch or a fence, or opening a door before walking through it, can all be described as Off-mesh links.

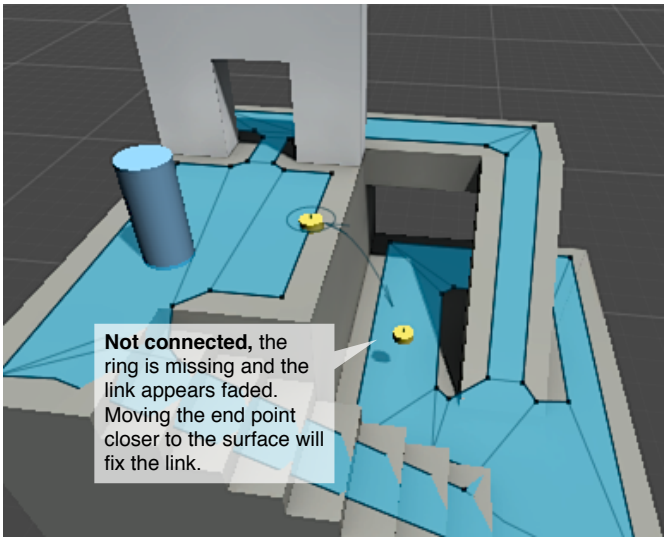
Properties



Property	Function
Start	Object describing the start location of the Off-Mesh Link.
End	Object describing the start location of the Off-Mesh Link.
Cost Override	If value is positive, use it when calculating path cost on processing a path request. Otherwise, the default cost is used (the cost of the area to which this game object belongs). If the Cost Override is set to the value 3.0, moving over the off-mesh link will be three times more expensive than moving the same distance on a default **NavMesh** area. The cost override becomes useful when you want to make the agents generally favor walking, but use the off-mesh link when the walk distance is clearly longer.
Bi-Directional	If enabled, the link can be traversed in either direction. Otherwise, it can only be traversed from *Start* to *End*.
Activated	Specifies if this link will used by the pathfinder (it will just be ignored if this is set to false).
Auto Update Positions	When enabled, the Off-Mesh link will be reconnected to the NavMesh when the end points move. If disabled the link will stay at its start location even if the end points are moved.
Navigation Area	Describes the navigation area type of the link. The area type allows you to apply a common traversal cost to similar area types and prevent certain characters from accessing the Off-Mesh Link based on the agent's Area Mask.

Details





If the agent does not traverse an Off-Mesh link make sure that both end points are connected correctly. A properly connected end point should show a circle around the access point.

Another common cause is that the Navmesh Agent's *Area Mask* does not have the Off-Mesh Link's area included.

Further Reading

Creating an Off-Mesh Link – workflow for setting up and off-mesh link.
Building Off-Mesh Links Automatically - how to automatically create.
Off-Mesh Link scripting reference - full description of the Off-Mesh Link scripting API.

Navigation How-Tos

Leave feedback

This section provides a set of techniques and code samples to implement common tasks in navigation. As with all code in our documentation, you are free to use it for any purpose without crediting Unity.

Telling a NavMeshAgent to Move to a Destination

Leave feedback

You can tell an agent to start calculating a path simply by setting the NavMeshAgent.destination property with the point you want the agent to move to. As soon as the calculation is finished, the agent will automatically move along the path until it reaches its destination. The following code implements a simple class that uses a **GameObject** to mark the target point which gets assigned to the *destination* property in the *Start* function. Note that the script assumes you have already added and configured the NavMeshAgent **component** from the editor.

```
// MoveDestination.cs
using UnityEngine;

public class MoveDestination : MonoBehaviour {

   public Transform goal;

   void Start () {
      NavMeshAgent agent = GetComponent<NavMeshAgent>();
      agent.destination = goal.position;
   }
}
```

```
// MoveDestination.js
var goal: Transform;

function Start() {
  var agent: NavMeshAgent = GetComponent.<NavMeshAgent>();
  agent.destination = goal.position;
}
```

Moving an Agent to a Position Clicked by the Mouse

Leave feedback

This script lets you choose the destination point on the **NavMesh** by clicking the mouse on the object's surface. The position of the click is determined by a *raycast*, rather like pointing a laser beam at the object to see where it hits (see the page Rays from the Camera for a full description of this technique). Since the GetComponent function is fairly slow to execute, the script stores its result in a variable during the *Start* function rather than call it repeatedly in *Update*.

```
// MoveToClickPoint.cs
using UnityEngine;
using UnityEngine.AI;

public class MoveToClickPoint : MonoBehaviour {
    NavMeshAgent agent;

    void Start() {
        agent = GetComponent<NavMeshAgent>();
    }

    void Update() {
        if (Input.GetMouseButtonDown(0)) {
            RaycastHit hit;

            if (Physics.Raycast(Camera.main.ScreenPointToRay(Input.mousePosi
                agent.destination = hit.point;
            }
        }
    }
}
```

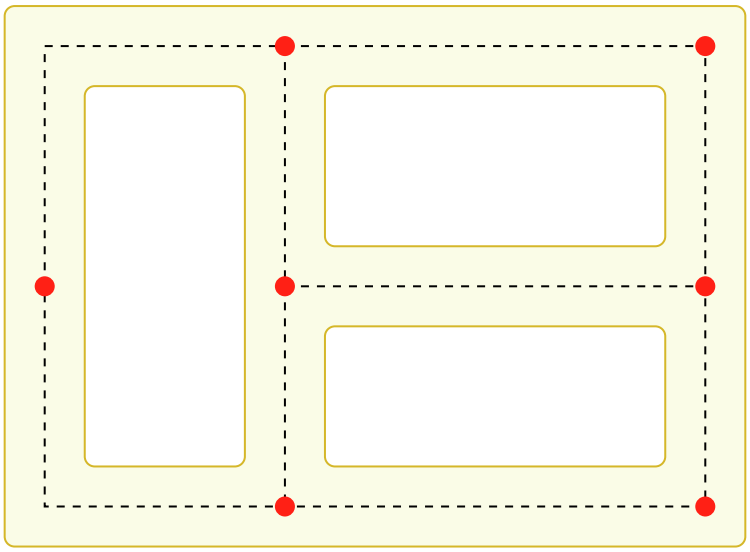
```
//MoveToClickPoint.js
var agent: NavMeshAgent;

function Start() {
    agent = GetComponent.<NavMeshAgent>();
}

function Update() {
    if (Input.GetMouseButtonDown(0)) {
        var hit: RaycastHit;
```

```
            if (Physics.Raycast(Camera.main.ScreenPointToRay(Input.mousePosition
                agent.destination = hit.point;
            }
        }
    }
```

Making an Agent Patrol Between a Set of Points

Leave feedback

Many games feature NPCs that patrol automatically around the playing area. The navigation system can be used to implement this behaviour but it is slightly more involved than standard pathfinding - merely using the shortest path between two points makes for a limited and predictable patrol route. You can get a more convincing patrol pattern by keeping a set of key points that are “useful” for the NPC to pass through and visiting them in some kind of sequence. For example, in a maze, you might place the key patrol points at junctions and corners to ensure the agent checks every corridor. For an office building, the key points might be the individual offices and other rooms.



A maze with key patrol points marked

The ideal sequence of patrol points will depend on the way you want the NPCs to behave. For example, a robot would probably just visit the points in a methodical order while a human guard might try to catch the player out by using a more random pattern. The simple behaviour of the robot can be implemented using the code shown below.

The patrol points are supplied to the script using a public array of Transforms. This array can be assigned from the **inspector** using **GameObjects** to mark the points’ positions. The *GotoNextPoint* function sets the destination point for the agent (which also starts it moving) and then selects the new destination that will be used on the next call. As it stands, the code cycles through the points in the sequence they occur in the array but you can easily modify this, say by using Random.Range to choose an array index at random.

In the *Update* function, the script checks how close the agent is to the destination using the remainingDistance property. When this distance is very small, a call to *GotoNextPoint* is made to start the next leg of the patrol.

```
// Patrol.cs
using UnityEngine;
using UnityEngine.AI;
using System.Collections;


public class Patrol : MonoBehaviour {
```

```
    public Transform[] points;
    private int destPoint = 0;
    private NavMeshAgent agent;


    void Start () {
        agent = GetComponent<NavMeshAgent>();

        // Disabling auto-braking allows for continuous movement
        // between points (ie, the agent doesn't slow down as it
        // approaches a destination point).
        agent.autoBraking = false;

        GotoNextPoint();
    }


    void GotoNextPoint() {
        // Returns if no points have been set up
        if (points.Length == 0)
            return;

        // Set the agent to go to the currently selected destination.
        agent.destination = points[destPoint].position;

        // Choose the next point in the array as the destination,
        // cycling to the start if necessary.
        destPoint = (destPoint + 1) % points.Length;
    }


    void Update () {
        // Choose the next destination point when the agent gets
        // close to the current one.
        if (!agent.pathPending && agent.remainingDistance < 0.5f)
            GotoNextPoint();
    }
}
```

```
// Patrol.js
var points: Transform[];
var destPoint: int = 0;
var agent: NavMeshAgent;
```

```
function Start() {
    agent = GetComponent.<NavMeshAgent>();

    // Disabling auto-braking allows for continuous movement
    // between points (ie, the agent doesn't slow down as it
    // approaches a destination point).
    agent.autoBraking = false;

    GotoNextPoint();
}


function GotoNextPoint() {
    // Returns if no points have been set up
    if (points.Length == 0)
        return;

    // Set the agent to go to the currently selected destination.
    agent.destination = points[destPoint].position;

    // Choose the next point in the array as the destination,
    // cycling to the start if necessary.
    destPoint = (destPoint + 1) % points.Length;
}


function Update() {
    // Choose the next destination point when the agent gets
    // close to the current one.
    if (!agent.pathPending && agent.remainingDistance < 0.5f)
        GotoNextPoint();
}
```

Coupling Animation and Navigation

Leave feedback

The goal of this document is to guide you to setup navigating humanoid characters to move using the navigation system.

We'll be using Unity's built-in systems for animation and navigation along with custom scripting to achieve this.

It's assumed you're familiar with the basics of Unity and the Mecanim animation system.

An example project is available — so you don't have add **scripts** or set up animations and animation controller from scratch:

NavigationAnimation_53.zip *Works with Unity 5.3+*

Creating the Animation Controller

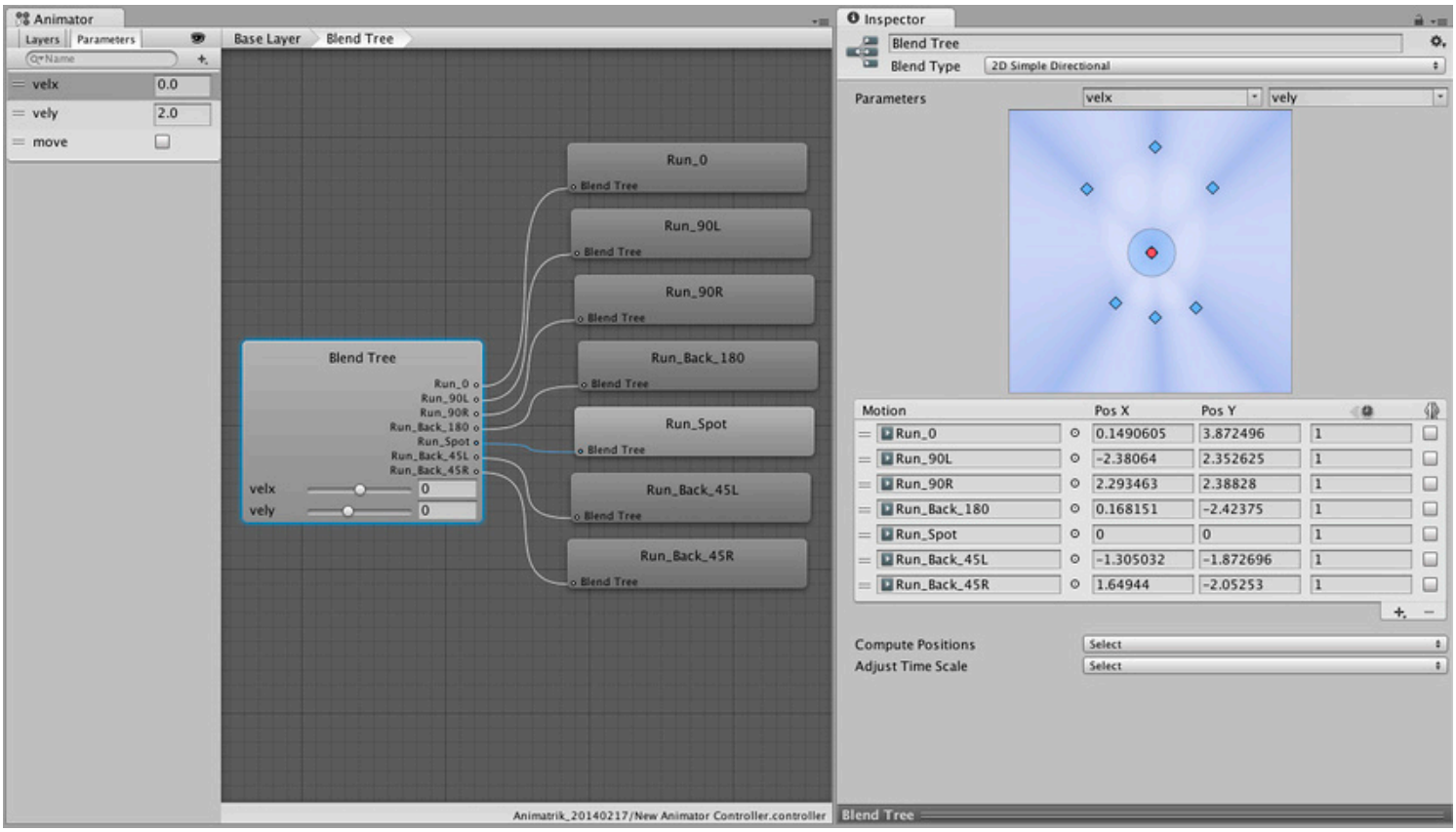
To get a responsive and versatile animation controller — covering a wide range of movements — we need a set of animations moving in different directions. This is sometimes referred to as a strafe-set.

In addition to the move animations we need an animation for the standing character.

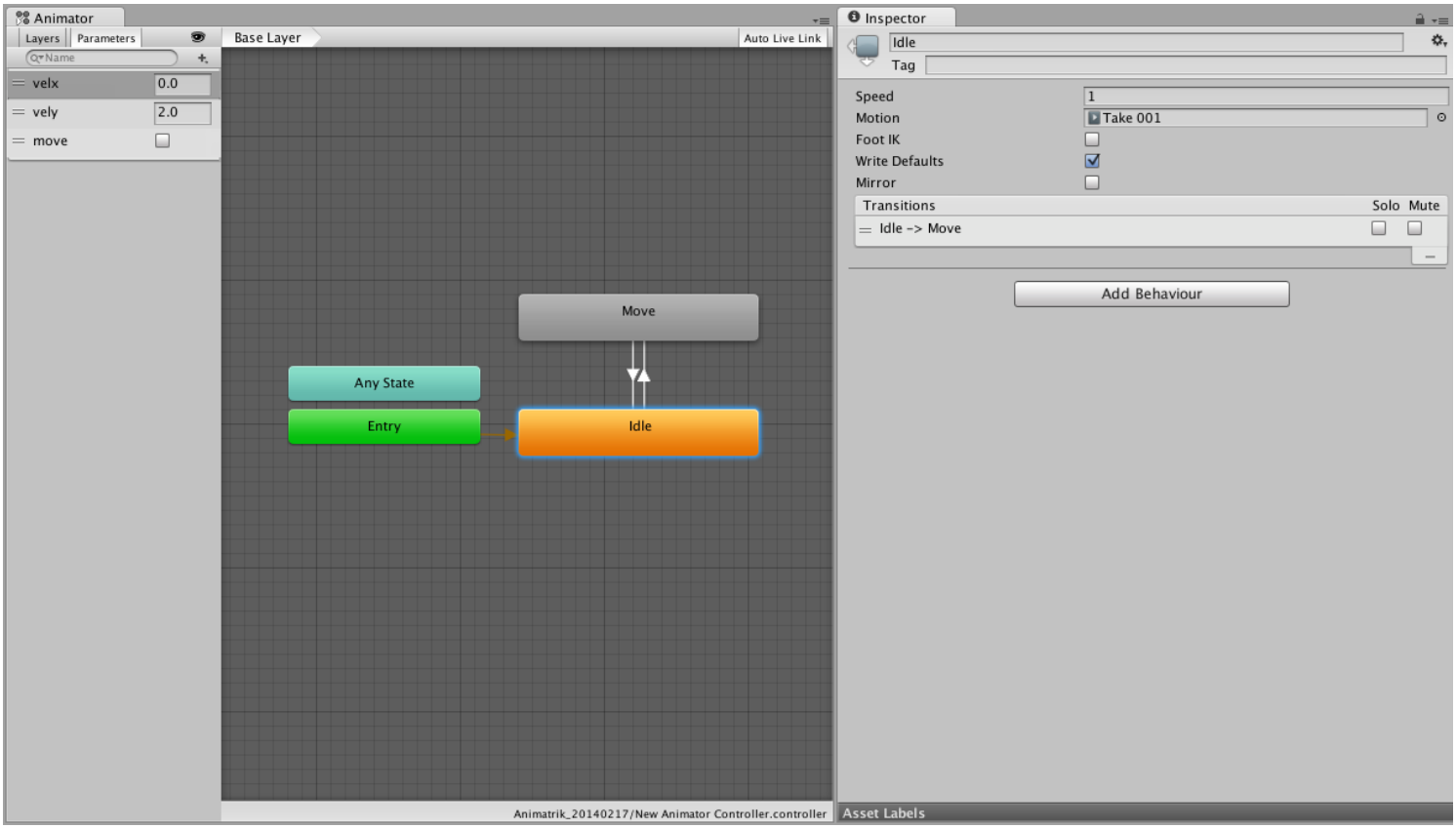
We proceed by arranging the strafe-set in a 2D blend tree — choose blend type: **2D Simple Directional** and place animations using **Compute Positions > Velocity XZ**

For blending control we add two float parameters **velx** and **vely**, and assign them to the blend tree.

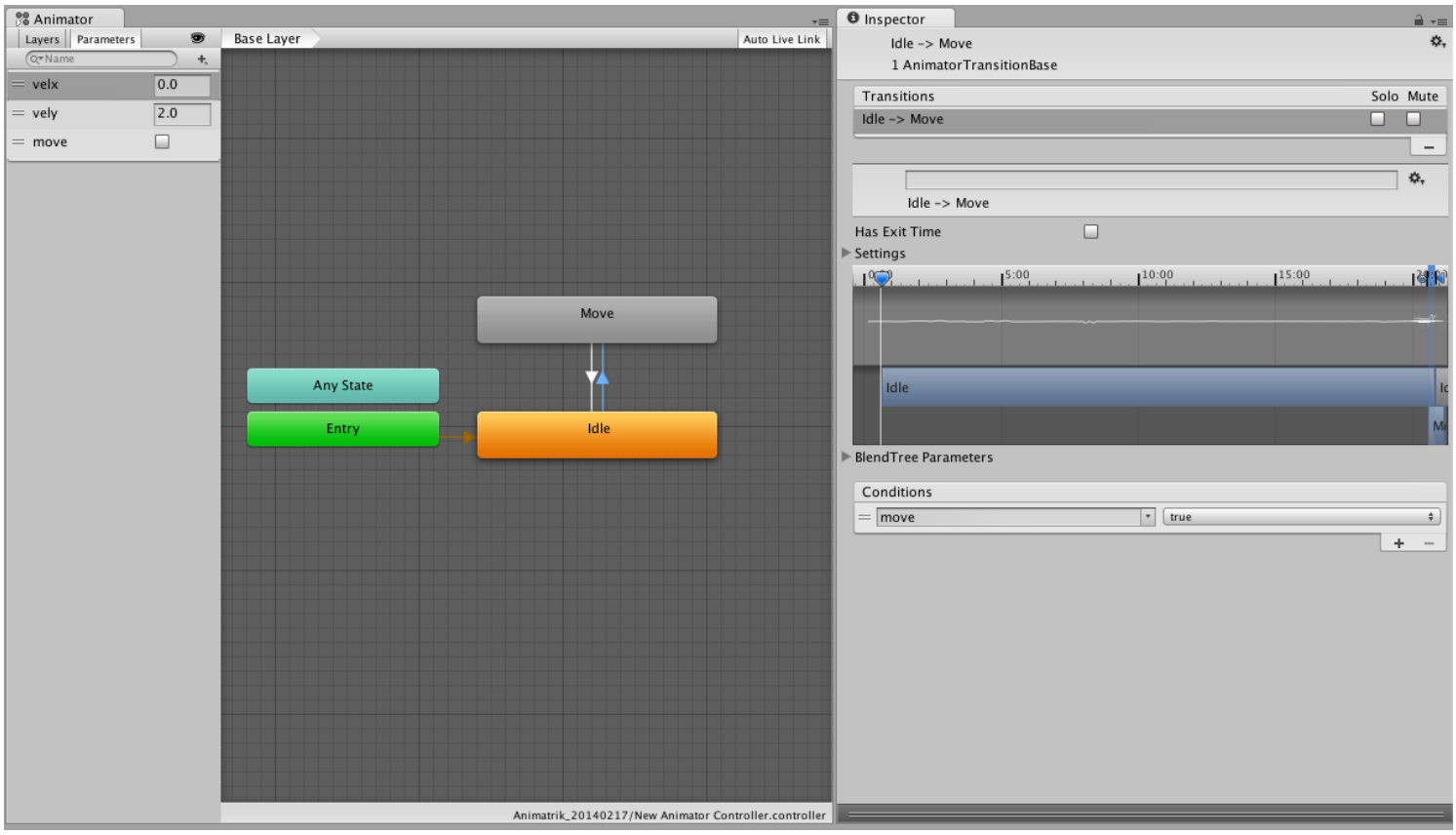
Here we'll be placing 7 run animations — each with a different velocity. In addition to the forwards (+ left/right) and backwards (+ left/right) we also use an **animation clip** for running on the spot. The latter is highlighted in the center of the 2D blend map below. The reason for having an animation running on the spot is two-fold, firstly it preserves the style of running when blended with the other animations. Secondly the animation prevents foot-sliding when blending.



Then we add the idle animation clip in it's own node (**Idle**). We now have two discrete animation states that we couple with 2 transitions.



To control the switch between the moving and idle states we add a boolean control parameter **move**. Then disable the **Has Exit Time** property on the transitions. This allows the transition to trigger at any time during the animation. Transition time should be set to around 0.10 second to get a responsive transition.



Now place the new created animation controller on the character you want to move.

Press play and select the character in the **Hierarchy window**. You can now manually control the animation values in the **Animator window** and change the move state and velocity.

The next step is to create other means of controlling the **animation parameters**.

Navigation Control

Place a **NavMeshAgent component** on the character and adjust the radius, height and to match the character - additionally change the speed property to match the maximum speed in the **animation blend tree**.

Create a **navmesh** for the **scene** you've placed the character in.

Next we need to tell the character where to navigate to. This typically is very specific to the application. Here we choose a click to move behavior — the character moved to the point in the world where the user has clicked on the screen.

```
// ClickToMove.cs
using UnityEngine;
using UnityEngine.AI;

[RequireComponent (typeof (NavMeshAgent))]
public class ClickToMove : MonoBehaviour {
    RaycastHit hitInfo = new RaycastHit();
    NavMeshAgent agent;
```

```
    void Start () {
        agent = GetComponent<NavMeshAgent> ();
    }
    void Update () {
        if(Input.GetMouseButtonDown(0)) {
            Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
            if (Physics.Raycast(ray.origin, ray.direction, out hitInfo))
                agent.destination = hitInfo.point;
        }
    }
}
```

Pressing play now — and clicking around in the scene — you'll see the character move around in the scene. However — the animations don't match the movement at all. We need to communicate the state and velocity of the agent to the animation controller.

To transfer the velocity and state info from the agent to the animation controller we will add another script.

```
// LocomotionSimpleAgent.cs
using UnityEngine;
using UnityEngine.AI;

[RequireComponent (typeof (NavMeshAgent))]
[RequireComponent (typeof (Animator))]
public class LocomotionSimpleAgent : MonoBehaviour {
    Animator anim;
    NavMeshAgent agent;
    Vector2 smoothDeltaPosition = Vector2.zero;
    Vector2 velocity = Vector2.zero;

    void Start ()
    {
        anim = GetComponent<Animator> ();
        agent = GetComponent<NavMeshAgent> ();
        // Don't update position automatically
        agent.updatePosition = false;
    }

    void Update ()
    {
        Vector3 worldDeltaPosition = agent.nextPosition - transform.position;

        // Map 'worldDeltaPosition' to local space
```

```
        float dx = Vector3.Dot (transform.right, worldDeltaPosition);
        float dy = Vector3.Dot (transform.forward, worldDeltaPosition);
        Vector2 deltaPosition = new Vector2 (dx, dy);

        // Low-pass filter the deltaMove
        float smooth = Mathf.Min(1.0f, Time.deltaTime/0.15f);
        smoothDeltaPosition = Vector2.Lerp (smoothDeltaPosition, deltaPosition, s

        // Update velocity if time advances
        if (Time.deltaTime > 1e-5f)
            velocity = smoothDeltaPosition / Time.deltaTime;

        bool shouldMove = velocity.magnitude > 0.5f && agent.remainingDistance >

        // Update animation parameters
        anim.SetBool("move", shouldMove);
        anim.SetFloat ("velx", velocity.x);
        anim.SetFloat ("vely", velocity.y);

        GetComponent<LookAt>().lookAtTargetPosition = agent.steeringTarget + tra
    }

    void OnAnimatorMove ()
    {
        // Update position to agent position
        transform.position = agent.nextPosition;
    }
}
```

This script deserves a little explanation. It's placed on the character — which has an **Animator** and a **NavMeshAgent** component attached — as well as the click to move script above.

First the script tells the agent not to update the character position automatically. We handle the position update that last in the script. The orientation is updated by the agent.

The animation blend is controlled by reading the agent velocity. It is transformed into a relative velocity (based on character orientation) — and then smoothed. The transformed horizontal velocity components are then passed to the **Animator** and additionally the state switching between idle and moving is controlled by the speed (i.e. velocity magnitude).

In the `OnAnimatorMove()` callback we update the position of the character to match the **NavMeshAgent**.

Playing the scene again gives show that animation matches the movement to as close as possible.

Improving the Quality of the Navigating Character

To improve the quality of the animated and navigating character we will explore a couple of options.

Looking

Having the character to look and turn towards points of interest is important to convey attention and anticipation. We'll use the animation systems lookat API. This calls for another script.

```
// LookAt.cs
using UnityEngine;
using System.Collections;

[RequireComponent (typeof (Animator))]
public class LookAt : MonoBehaviour {
    public Transform head = null;
    public Vector3 lookAtTargetPosition;
    public float lookAtCoolTime = 0.2f;
    public float lookAtHeatTime = 0.2f;
    public bool looking = true;

    private Vector3 lookAtPosition;
    private Animator animator;
    private float lookAtWeight = 0.0f;

    void Start ()
    {
        if (!head)
        {
            Debug.LogError("No head transform - LookAt disabled");
            enabled = false;
            return;
        }
        animator = GetComponent<Animator> ();
        lookAtTargetPosition = head.position + transform.forward;
        lookAtPosition = lookAtTargetPosition;
    }

    void OnAnimatorIK ()
    {
        lookAtTargetPosition.y = head.position.y;
        float lookAtTargetWeight = looking ? 1.0f : 0.0f;

        Vector3 curDir = lookAtPosition - head.position;
        Vector3 futDir = lookAtTargetPosition - head.position;

        curDir = Vector3.RotateTowards(curDir, futDir, 6.28f*Time.deltaTime, flo
        lookAtPosition = head.position + curDir;
```

```
        float blendTime = lookAtTargetWeight > lookAtWeight ? lookAtHeatTime : l
        lookAtWeight = Mathf.MoveTowards (lookAtWeight, lookAtTargetWeight, Time
        animator.SetLookAtWeight (lookAtWeight, 0.2f, 0.5f, 0.7f, 0.5f);
        animator.SetLookAtPosition (lookAtPosition);
    }
}
```

Add the script to the character and assign the head property to the head transform in your characters transform hierarchy. The LookAt script has no notion of navigation control — so to control where to look we go back to the **LocomotionSimpleAgent.cs** script and add a couple of lines to control the looking. Add the end of `Update()` add:

```
        LookAt lookAt = GetComponent<LookAt> ();
        if (lookAt)
            lookAt.lookAtTargetPosition = agent.steeringTarget + transform.forwa
```

This will tell the **LookAt** script to set the point of interest to approximately the next corner along the path or — if no corners — to the end of the path.

Try it out.

Animation Driven Character using Navigation

The character has so far been controlled completely by the position dictated by the agent. This ensures that the avoidance of other characters and obstacles translates directly to the character position. However it may lead to foot-sliding if the animation doesn't cover the proposed velocity. Here we'll relax the constraint of the character a bit. Basically we'll be trading the avoidance quality for animation quality.

Replace the `OnAnimatorMove()` callback on the **LocomotionSimpleAgent.cs** script replace the line with the following

```
    void OnAnimatorMove ()
    {
        // Update position based on animation movement using navigation surface l
        Vector3 position = anim.rootPosition;
        position.y = agent.nextPosition.y;
        transform.position = position;
    }
```

When trying this out you may notice the that character can now drift away from the agent position (green wireframe cylinder) . You may need to limit that character animation drift. This can be done either by pulling the agent towards the character — or pull the character towards the agent position. Add the following at the end of the `Update()` method on the script **LocomotionSimpleAgent.cs**.

```
        // Pull character towards agent
        if (worldDeltaPosition.magnitude > agent.radius)
            transform.position = agent.nextPosition - 0.9f*worldDeltaPosition;
```

Or — if you want the agent to follow the character.

```
        // Pull agent towards character
        if (worldDeltaPosition.magnitude > agent.radius)
            agent.nextPosition = transform.position + 0.9f*worldDeltaPosition;
```

What works best very much depends on the specific use-case.

Conclusion

We have set up a character that moves using the navigation system and animates accordingly. Tweaking the numbers of blend time, look-at weights **etc**. can improve the looks — and is a good way to further explore this setup.

Unity Services

Leave feedback

Unity is more than an engine. It also brings a growing range of integrated services to engage, retain and monetize audiences. Unity provides a growing range of complimentary services to help developers make games and engage, retain and monetize audiences.

Our Unity Ads, Unity **Analytics**, **Unity Cloud Build** and Unity Multiplayer are fully integrated with the Unity Editor to make creating and managing games as smooth, simple and rewarding an experience as possible.

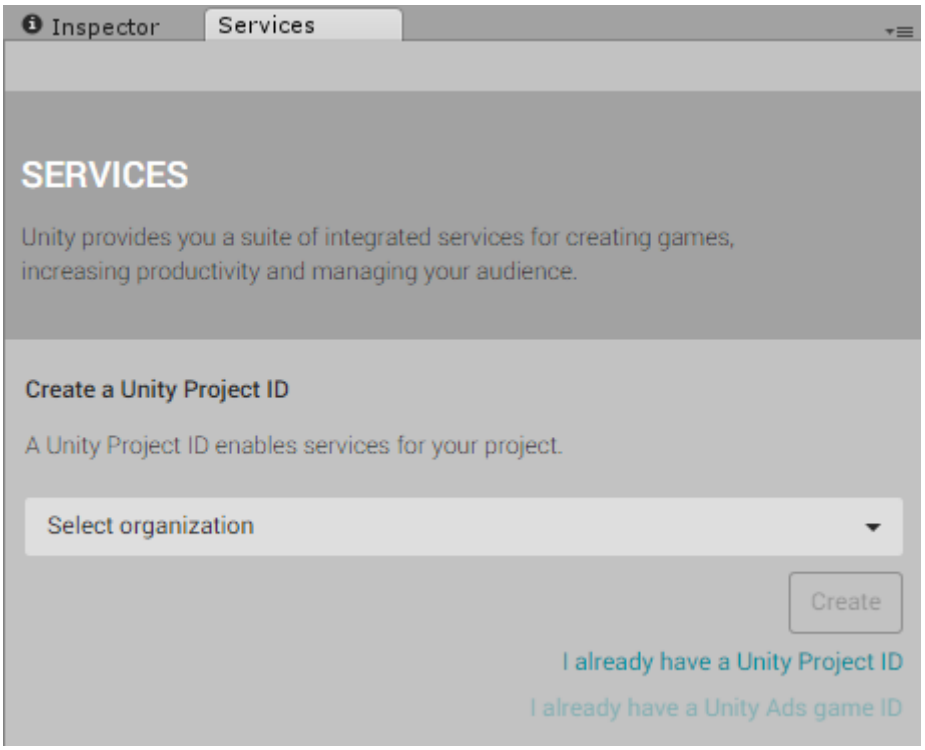
See the Knowledge Base Services section for tips, tricks and troubleshooting.

Setting up your project for Unity Services

Leave feedback

To get started with Unity's family of services, you must first link your project to a **Unity Services Project ID**. A Unity Services Project ID is an online identifier which is used across all Unity Services. These can be created within the Services window itself, or online on the Unity Services website. The simplest way is to use the Services window within Unity, as follows:

To open the Services Window, go to **Window** > **General** > **Services**, or click the cloud button in the toolbar. . If you have not yet linked your project with a Services ID, the following appears:



The Services window

This allows you to create a new Project ID or select an existing one.

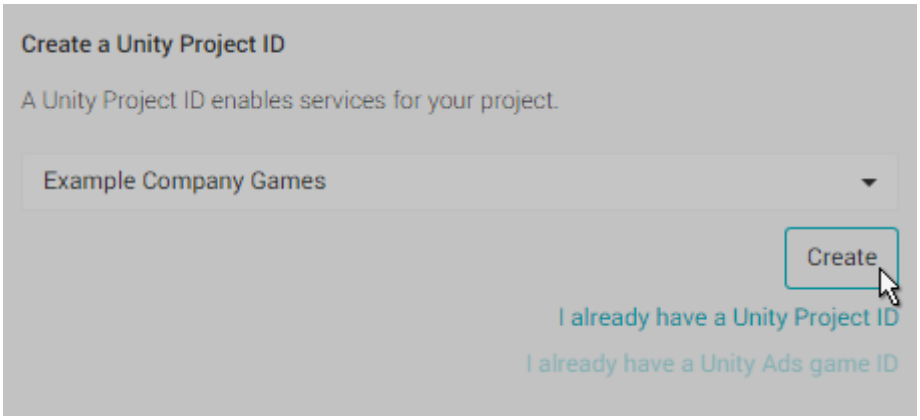
To create a Project ID, you must specify an **Organization** and a **Project Name**.

If this is the first time you are using any of the Unity Services, you will need to select both an Organization and a Project name. The **Select organization** field is typically your company name.

When you first open a Unity Account (usually when you first download and install Unity), a default "Organization" is created under your account. This default Organization has the same name as your account username. If this is the Organization you want your Project to be associated with, select this organization from the list.

Sometimes people need to be able to work in more than one organization - for example, if you are working with different teams on different projects. If you need to work with multiple organizations, you can add and manage your organizations on the organization ID web page. If you belong to multiple organizations, they will appear in the list.

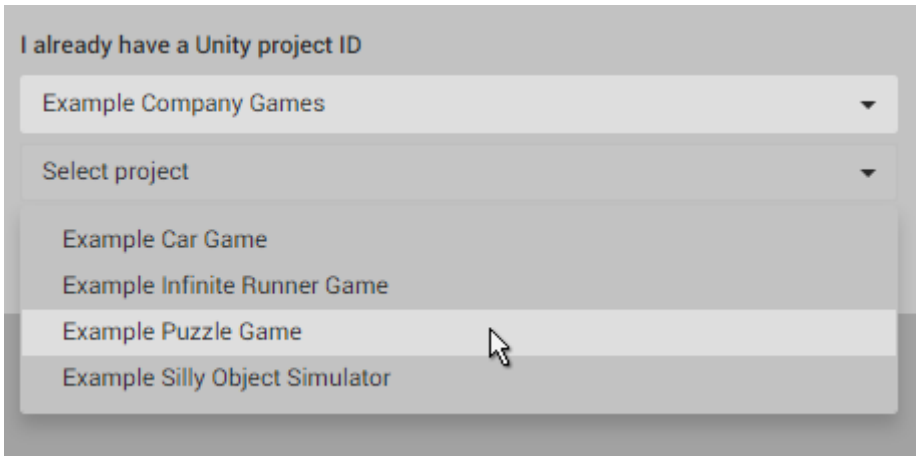
Select your organization and click **Create**.



Creating a new Unity Services Project ID

The Project name for your new Unity Services Project ID is automatically filled in with the name you picked for your Project when you first created it. You can change the Project name later in the **Settings** section of the **Services** window.

Sometimes you might need to re-link your project with the Unity Services Project ID. In this case, select the Project from the list of your existing Projects that are registed with Unity Services.



Selecting an existing project

Unity Organizations

Leave feedback

Unity Organizations are the containers for your Projects and subscriptions. If you are an individual game developer, you might not need to work with the features provided by an Organization. However, to work with other team members, you must understand how to use accounts, Organizations, subscriptions, and seats.

Organizations are one way in which a collection of users can collaborate on Unity projects. An Organization can consist of a single user, and expand to include multiple users at a later time. You can associate multiple Organizations with your account.

A subscription defines the software license level available to an Organization. There are three subscription tiers: Personal, Plus, or Pro. For more information on requirements for each of the Unity subscription tiers, see the **Tier Eligibility** section of Unity Pro, Unity Plus and Unity Personal Software Additional Terms.

A seat represents an individual license to use Unity. When purchasing a Pro or Plus subscription, select the number of seats to purchase in that tier and then assign the seats to users in your Organization. You can also assign seats at a Project level to users who are not members of your Organization.

2018–04–25 Page published with editorial review

Subscriptions and seats

Leave feedback

To use the Unity Editor and Services, you must create a Unity Developer Network (UDN) account.

When you create an account, you get:

A default Organization
A Unity Personal Tier subscription (license) in the Organization
An Editor seat for the license in the subscription
If you are not eligible to use a Unity Personal tier subscription, you must upgrade to either the Unity Plus or Pro tier. When you subscribe to Plus or Pro, you get:

A subscription (license) attached to an Organization of your choice to your account.
An Editor seat for the license in the subscription
When you purchase additional Subscriptions, you do so through an Organization. You can purchase additional subscriptions on the Unity ID dashboard (see Members and groups). If you are part of a company, this allows you to organize your licenses under a company Organization while keeping your other activities in Unity separate. For more information, see Managing your Organization.

Subscription seats

A subscription seat represents a single user license, and allows users to work together on Projects in the Editor. If your Organization uses a Pro or Plus subscription, all users working on your Organization's Projects must have an Editor seat at the same tier or higher. If a user has a lower tier subscription, you must assign them a seat from your license.

Note: You must be an Owner or Manager in the Organization to assign seats, see Organization roles.

To assign seats:

Sign in to the Unity ID dashboard.
In the left navigation bar, click **Organizations**.
Select the Organization.
In the left navigation pane, click **Subscriptions & Services**.
Select the subscription from which you are assigning the seat.
Click the **Manage seats** button, then select the Organization member that you want to assign a seat to.
Click the **Assign Seat(s)** button.
The selected member receives an email with information on how to activate Unity.

Assigning a seat gives the user access to Editor features at the Organization's subscription level. This means the user can have a Personal tier license but use an assigned seat to access a higher tier subscription license.

You can purchase additional seats for your subscription at any time on Unity's website. For information on activating Unity, see Online Activation.

Working with individuals outside of your Organization

If you want to collaborate with individuals outside your Organization without giving them access to your Organization's sensitive information, add the user directly to a specific Project. If the contributor has their own Plus or Pro Editor Seat that matches the Organization's subscription tier, you do not need to assign them one of yours.

To add a user to a specific Project:

Sign in to the Unity Services Dashboard.
Select the Project that you want to add a user to.
In the left navigation column, click **Settings**, then click **Users**.
In the **Add a person or group** field, enter the user's email address.
To allow users to access the Collaborate and Cloud Build features on your Project, you must assign them a Unity Teams seat, which is separate from the Editor seats associated with subscriptions. If the specified user does not have a Unity Teams seat, one is assigned by default. If you do not want the user to collaborate with Unity Teams, uncheck the **Also assign a Unity Teams Seat to this user** checkbox.

For more information on Unity Teams, see Working with Unity Teams.

2018–04–25 Page published with editorial review

Managing your Organization

Leave feedback

To create or manage Organizations, sign in to the Unity ID dashboard and select **Organizations** from the left navigation bar.

Creating Organizations

You can use Organizations to segregate your Projects by the entity for which they are being developed. You can create as many Organizations as you like. To keep your activities for different companies or personal use separate, use a seperate Organization for each entity.

Separating your activities through Organizations also facilitates seamless personnel changes. For example, if an employee leaves the company, you can reallocate their seat because the subscription is tied to the Organization instead of the user.

To create a new Organization:

Sign in to the Unity ID dashboard.
In the left navigation bar, click **Organizations**.
In the **Organizations** pane, click the **Add new** button.
In the **Organization Name** text field, enter the name of your Organization, and then click the **Create** button.
The following sections describe the information and tasks available for your Organizations from the Unity ID dashboard.

Members & Groups

With an Organization selected, click **Members & Groups** from the left navigation bar to view or add members and groups within that Organization, or assign user roles.

Adding users

To work with other users, you must give them access to your Project(s). To do this, either:

Add the user to your Organization. This allows the user to view all of the Projects within the Organization. If the user needs to modify information about your subscription, you can assign them an Owner or Manager role.
Or:

Add the user directly to an individual Project. This gives the user access to one Project, with no access to Organization and subscription information.
Adding a user to an Organization or a Project allows them to view information about a Project. For example, they can view Project **Analytics** and the results of Performance Reporting.

For the user to work on Projects, as opposed to just viewing information, they must have a seat tier that matches the Organization's subscription tier. If they do not have one of their own, you must assign them one. For more information, see Subscription Seats.

Organization roles

You can assign different administrative roles to members of your Organization. Each role grants access to certain functions within the Organization.

To assign a team member to a role:

Sign in to the Unity ID dashboard.
Click **Organizations** on the left navigation bar.The **Organizations** page contains a list of the Organization names associated to your account.
Click the cog icon next to the name of the Organization.
Click **Members & Groups** in the left navigation bar. This page contains all members associated with the Organization.
Find the name of the member whose role you want to change, and then click the pencil icon.
Choose the member's role from the drop-down menu titled **Organization Role**. Once you've selected the new role, click **Save**.
Note: Organization Managers can also assign member roles. However, they can only assign the role of User or Guest.

Owners can access all settings in any of their Organizations' subscriptions, across all Projects. Owners are the only users with access to payment instruments, invoices and billing data for the Organization.
Managers can access most settings in any of their Organizations' subscriptions, across all Projects. Managers can add users, access monetization data, and do almost everything an Owner can do. However, they cannot see billing and credit card information for the Organization.
Users can read and view all Organization and Services data, except Monetization data. They cannot edit Organization and Services data.
Guests are members of your Organization that have no permissions to view data within the Organization. The Guest role allows you to assign vendors and contractors a Teams Advanced seat in the Organization that allows them to use the Collaboration service. Guests only have permission to access the specific project to which they are assigned.
All Organization members can access all of the Organization's Projects.

To assign a member a different role for a specific Project:

Sign in to the Unity Services Dashboard.
Select the relevant Project.
In the left navigation column, click **Settings**, then click **Users**.
Click the **Add a person or group** field.
Select a Organization user from the displayed list or enter the email address of a user who has not been added to the Organisation.
If the specified user does not have a Unity Teams seat, Unity assigns them a seat automatically. If you do not want the user to collaborate on the Project using Unity Teams, uncheck the **Also assign a Unity Teams Seat to this user** checkbox. For more information on Unity Teams, see Working with Unity Teams.

Note: Because Services are enabled on a per Project basis, and each Project is tied to an Organization, only that Organization's Owner can enable and disable Services for its associated Projects. Regardless of your subscription tier or seat, you need certain roles or permissions within an Organization to use Services features or view related data on the Unity Services Dashboard.

Subscriptions & Services

With an Organization selected, click **Subscriptions & Services** from the left navigation bar to manage your Editor and Unity Teams subscriptions.

To view the status of a subscription, select the subscription you wish to view from the list. The subscription details page allows you to add and assign seats for that Subscription. It also allows you to extend or upgrade your subscriptions.

Service Usage

With an Organization selected, click **Service Usage** from the left navigation bar to view your Unity Teams cloud storage allocation and usage.

Unity Teams includes some cloud storage, which you can adjust on a month-to-month basis. Click **Manage storage** to add or subtract space. For more information on managing storage, see How Do I Get More Cloud Storage In the knowledge base.

Edit Organization

With an Organization selected, click **Edit Organization** from the left navigation bar to change the Organization's name and contact information. Only the Owner can change these settings.

Payment Methods

With an Organization selected, click **Payment Methods** from the left navigation bar to manage the credit cards that Unity stores for the Organization. Valid payment methods are required to purchase subscriptions or cloud storage.

Transaction History and Payouts

With an Organization selected, click **Transaction History** from the left navigation bar to view records of your purchases and payouts. Select a date range to query, then click **Apply** to view records for that period.

The **Purchases** tab reports purchases you've made during the specified period. The **Payouts** tab reports payments you received from Unity for ads revenue. To receive payouts, you must set up a payment profile by clicking **Payment Profile** in the left navigation bar. For more information on Unity Ads revenue, see Revenue and payment.

2018–04–25 Page published with editorial review

Managing your Organization's Projects

Leave feedback

When you create a new Project, it is assigned to an Organization. If you have not created any additional Organizations, it is automatically assigned to your default Organization. If multiple Organizations are available to your account, you can choose which one to use. As such, Projects inherit their Organization's role settings. However, if users need access to certain tools or information, you can give them different roles that are unique to each Project, without granting visibility across the entire Organization's portfolio of Projects. For more information, see Members & Groups in **Managing your Organization**.

As the Organization Owner, you also have the following tools to manage your Projects:

Archiving and restoring Projects
Changing Project names
Transferring Projects to a new Organization

Archiving and restoring Projects

Projects cannot be deleted which can lead to clutter on your dashboard. To reduce clutter, you can archive Projects you no longer use. Archiving a Project prevents it from displaying in your **Projects** list.

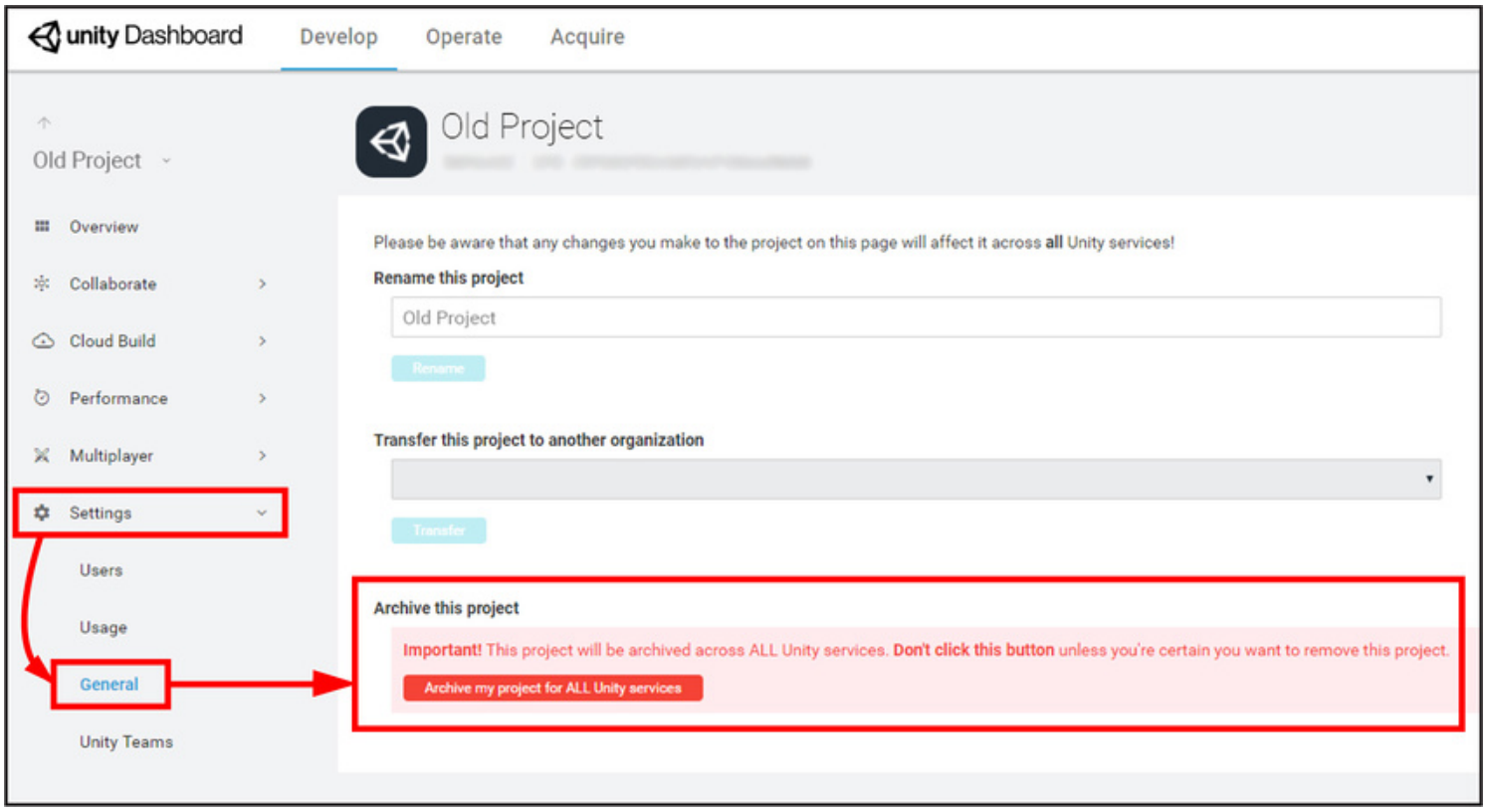
To archive a Project:

Sign in to the Unity Dashboard.

Select the Project to archive.

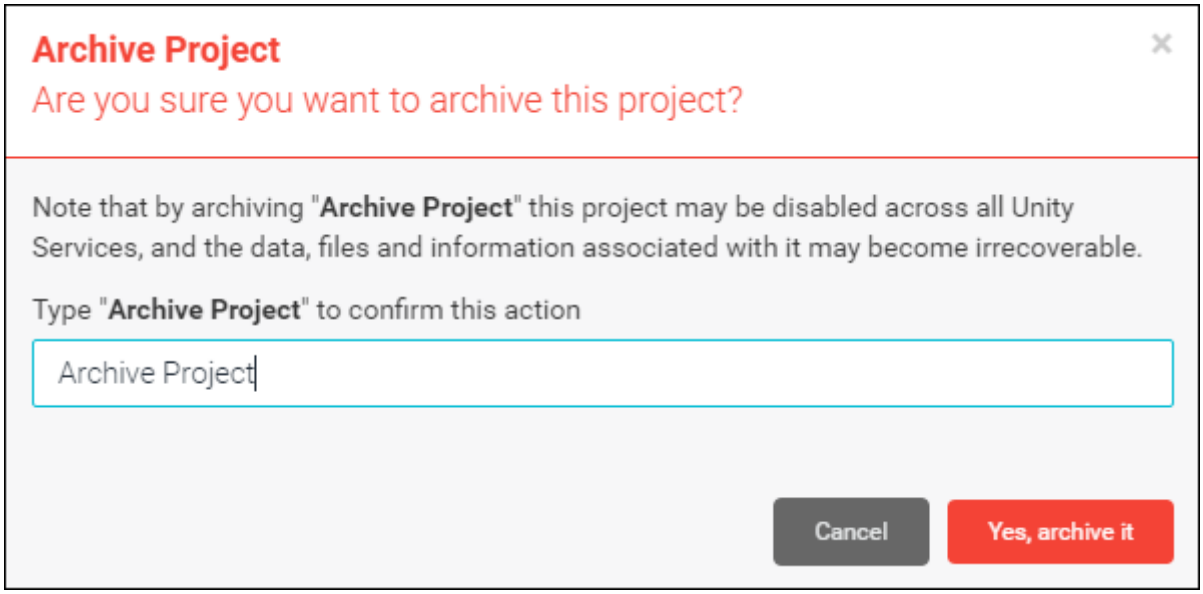
In the left navigation pane, click **Settings** > **General**.

In the Project pane, click the **Archive my project for ALL Unity services** button.



Archive Project window

In the Project archive pop-up window, enter the name of the Project to confirm that you want to archive the Project, then click the **Yes, archive it** button.

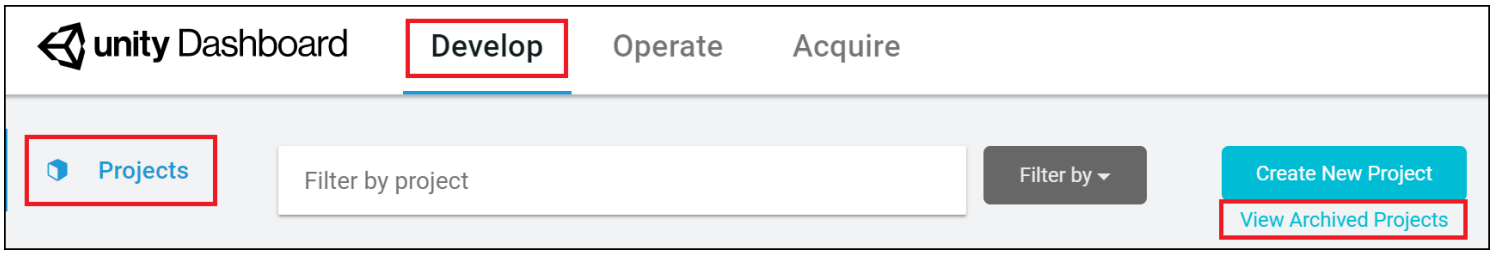


Archive Project confirmation window

To view your archived Projects and reactivate them:

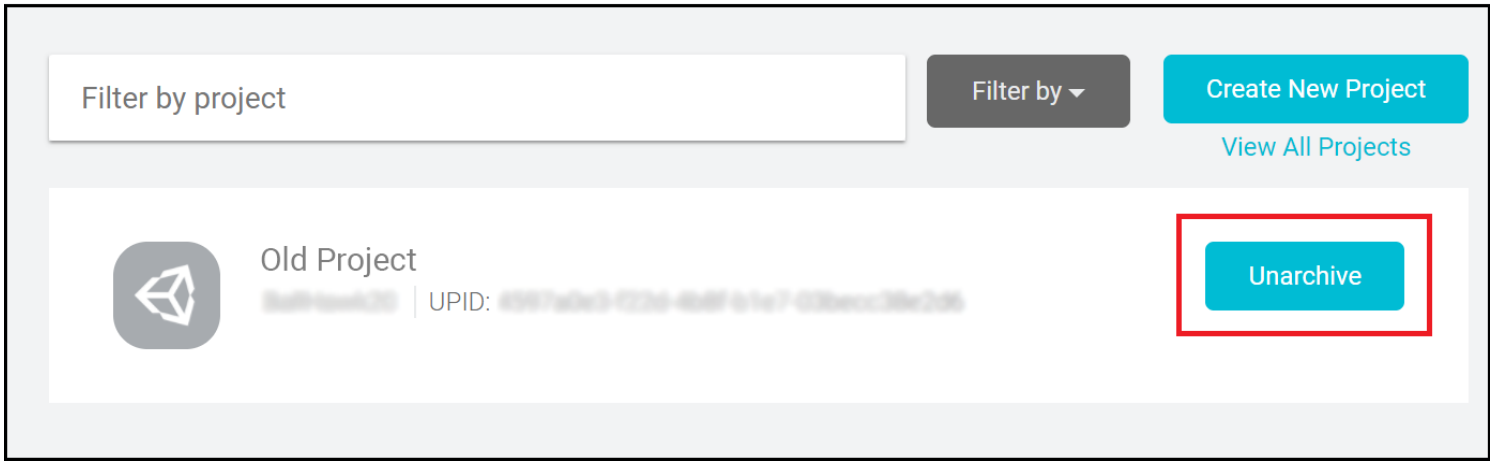
Sign in to the Unity Services Dashboard.

Click View Archived Projects on top right of the page.



View archived Projects window

Locate the Project to reactivate and the click the **Unarchive** button.



Unarchive Projects window

Changing the name of a Project

To change the name of a Project:

Sign in to the Unity Services Dashboard.
Select the Project to rename.
In the left navigation pane, click **Settings** > **General**.
In the **Rename this project** textbox, enter a new name for the Project, then click the **Rename** button.

2018–04–25 Page published with editorial review

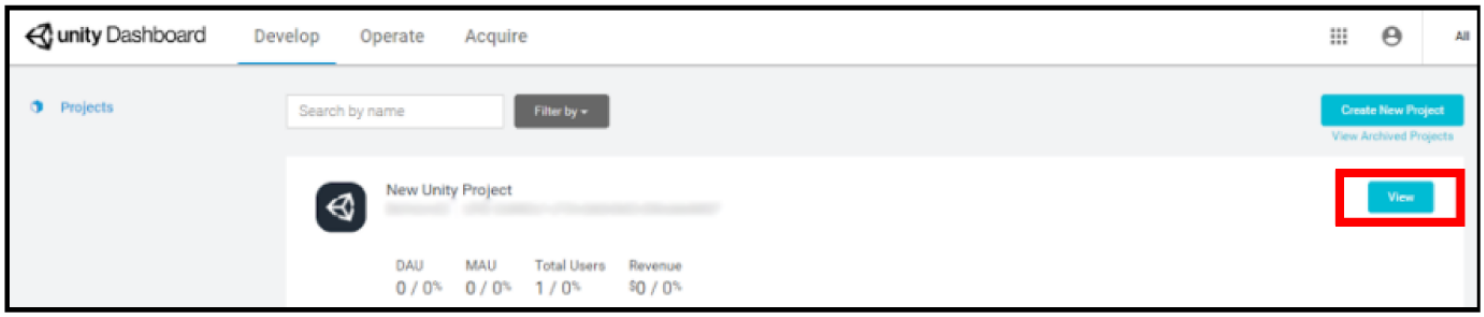
Transfer a Project to a new Organization

Leave feedback

To transfer a Unity Project from one organization to another, you must be an owner or manager of both organizations. You perform this operation through the Unity Developer Dashboard.

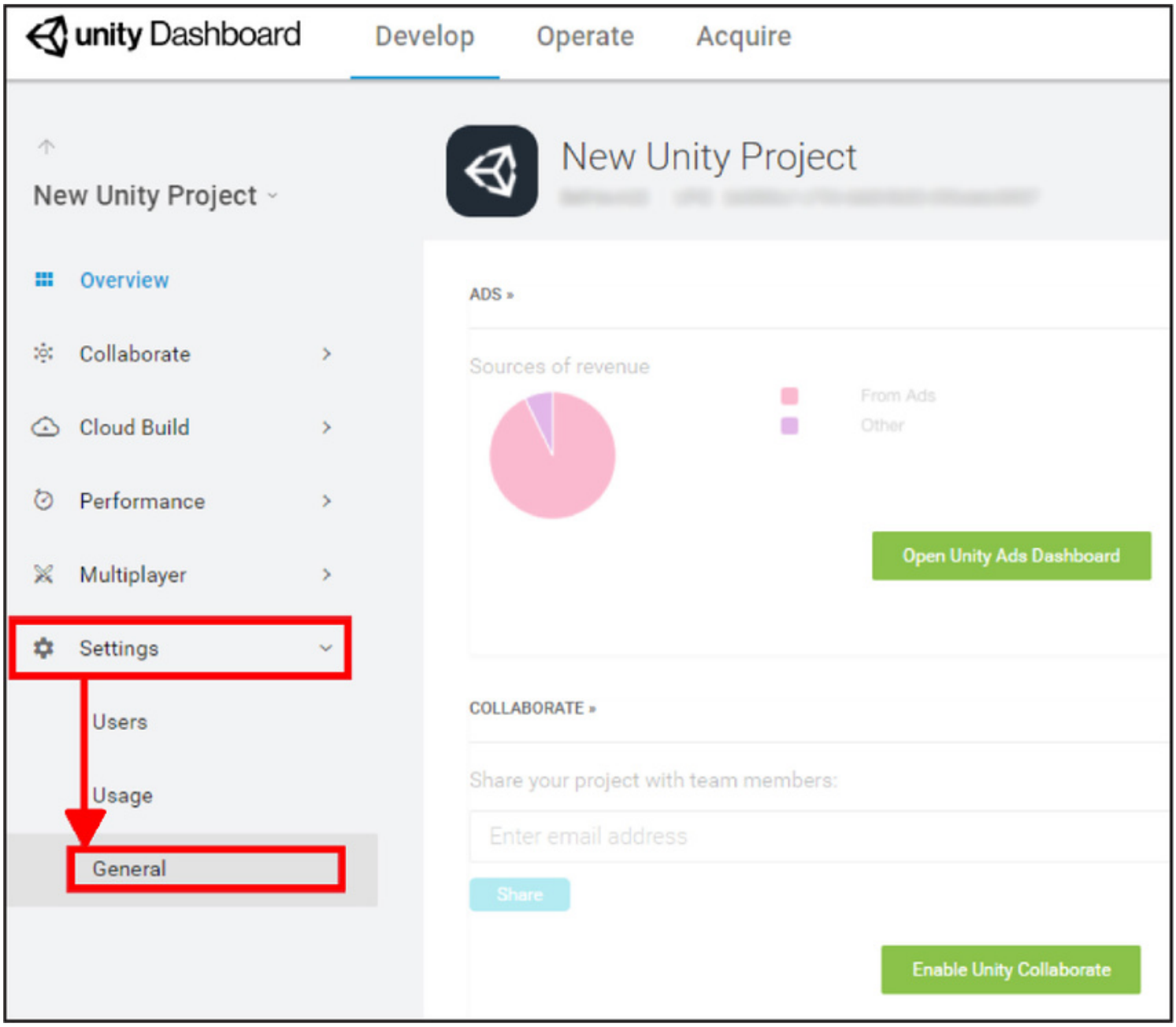
Note: You do not need to disable Unity Services for this process.

From the Developer Dashboard **Projects** list, locate the Project you want to transfer, and select **View**.



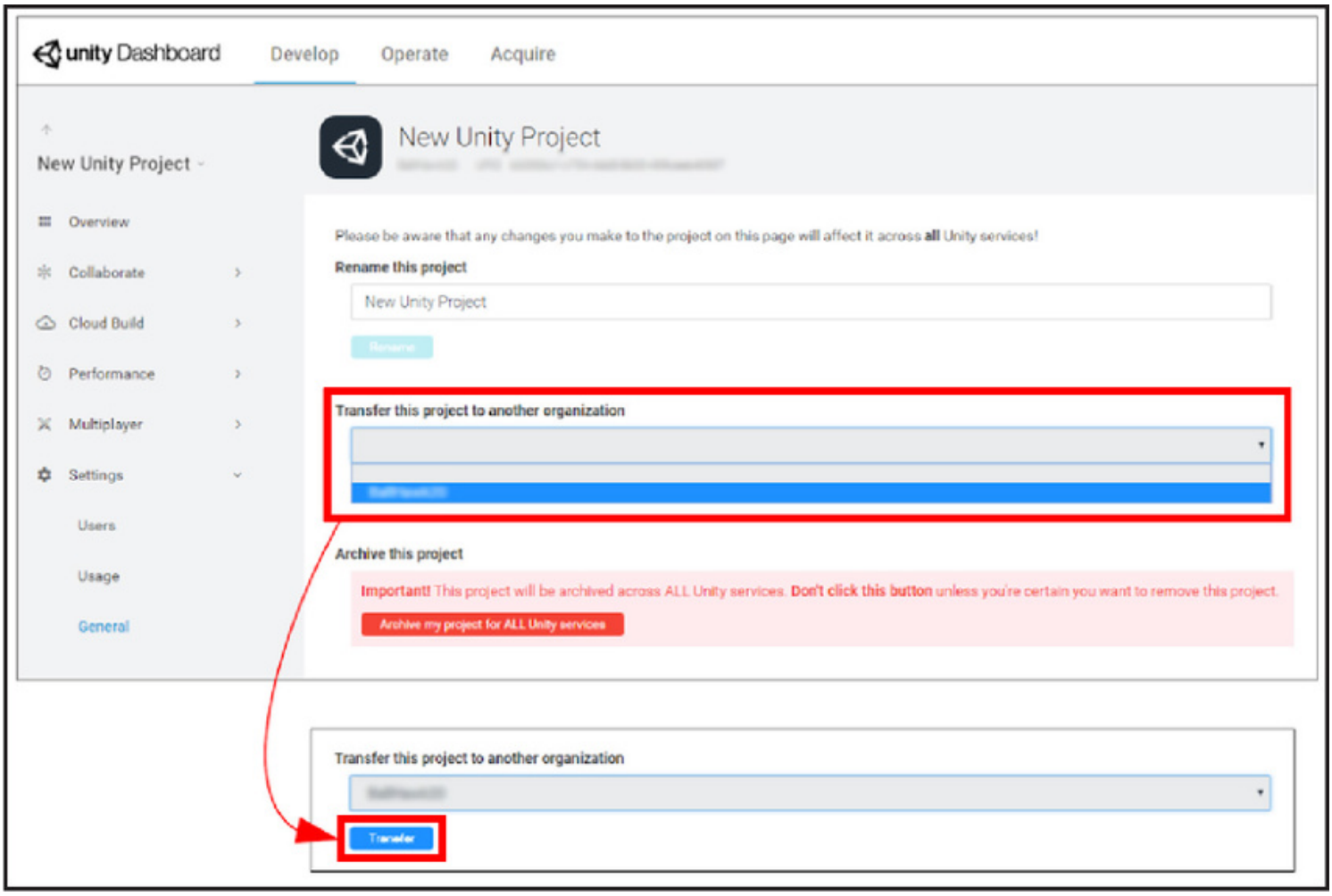
Selecting a Project on the Developer Dashboard

Select **Settings** on the left navigation bar, then select **General** from the drop-down options.



Selecting the General Settings for your Project

Select the **Transfer this project to another organization** drop-down, then select the destination organization from the list and click **Transfer**.



Transferring a Project to another organisation

A message appears indicating whether the transfer was successful. If you receive an error message, contact Unity Support.

2017–10–18 Page published with no editorial review

2017–10–18 - Project transfer service compatible with Unity 2017.1 onwards at this date; version compatibility may be subject to change.

Working with Unity Teams

Leave feedback

Unity Teams enables creative teams to work more efficiently together with features that enable collaboration and simplify workflows.

Unity Teams consists of the following services:

Unity Collaborate: Save, share, and sync your projects and use simple **version control** and cloud storage, all seamlessly integrated with Unity. For more information, see Unity Collaborate.
Unity Cloud Build: Automatically share builds with team members. If you subscribe to Teams Advanced you can build your Project in the cloud. For more information, see Unity Cloud Build.
Every Unity Project is associated with an Organization. Organizations are the container for your Unity subscription services.

When you need to purchase additional Editor seats, you create a subscription under an Organization and add seat licenses under it. When you assign an Editor seat to a Project user, it is assigned from one of your subscriptions.

Team seats are attached directly to an Organization. When you create your first Project, you get a Teams Basic subscription that has three Team seats. If you subscribe to Teams Advanced, the seats you buy are added to the Organization.

For Project team member to use Unity Teams, you must assign them a Team seat in your organization. You might also need to assign them an Editor seat if their license level does not match the license level of your Subscription.

When working with Unity Teams seats, remember that:

Unity Teams seats are separate from Editor seats.
Your Unity Teams seats are attached to your Organization, not a subscription under the Organization.
Subscribe to Unity Teams Advanced if you need to:

Collaborate with more than two other people
Use the full features of Cloud Build
Use more cloud storage
Get an extensive version history for Projects on which you collaborate
For more information on assigning a Unity Teams seat user, see Adding team members to your Unity Project.

2018–04–25 Page published with editorial review

Unity Ads

Leave feedback

Unity Ads is a mobile advertising network for **iOS** & Android that delivers market leading revenue from your user base, while increasing engagement and retention.

We are currently working on consolidating the documentation for Unity Ads into this user manual. In the meantime, the best place to get started and learn about using Unity Ads is the Unity Ads Knowledge Base.

Useful links

Integration Guide for Unity - A guide covering both methods for integrating Unity Ads in the Unity engine.

Unity Ads Admin panel - Here you can see reports, configure advanced monetization settings for your games, and perform other ads-related management tasks.

Unity Ads forums - get help and find answers to commonly asked questions.

Blog posts:

Showing ads in your game to make money

A Designer's Guide To Using Video Ads

2017–07–27 Page amended with editorial review

Unity Editor integration

Leave feedback

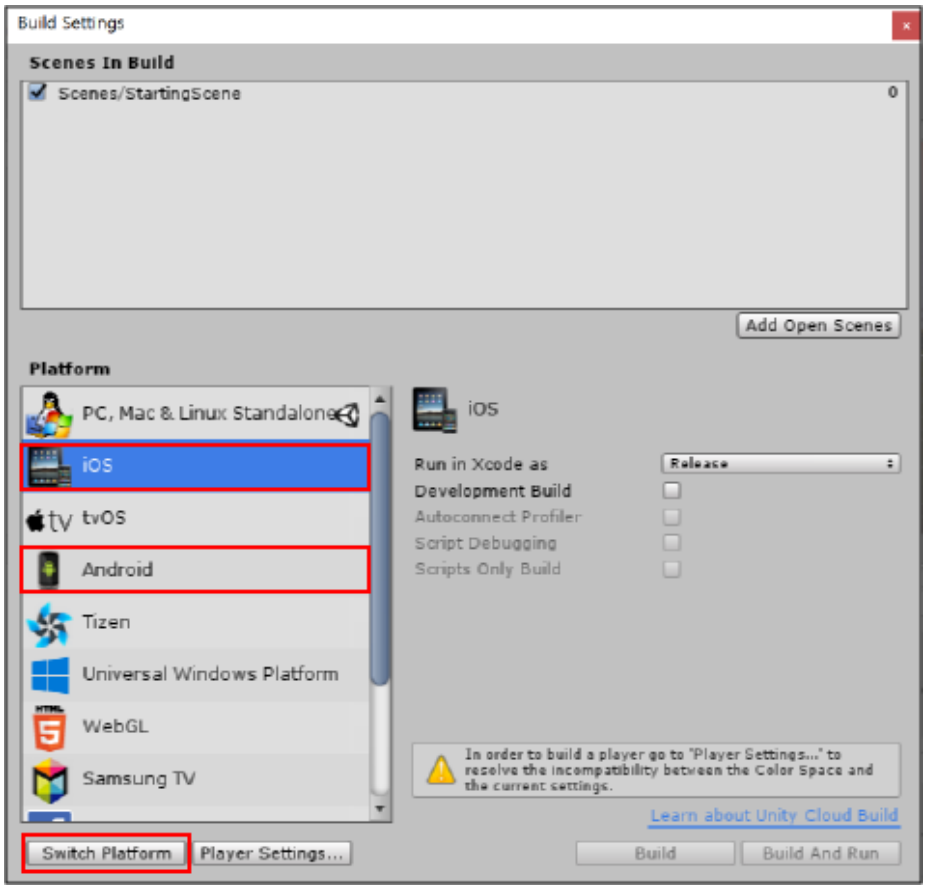
This guide covers both methods for integrating Unity Ads in the Unity engine:

Services Window integration
Asset Package integration

Setting build targets

Configure your project for a supported platform using the Build Settings window.

Set the platform to **iOS** or **Android**, then click **Switch Platform**.



The Build Settings window

Enabling the Ads Service

This process varies slightly depending on your integration preference; the Services Window method or the **Asset Package** method.

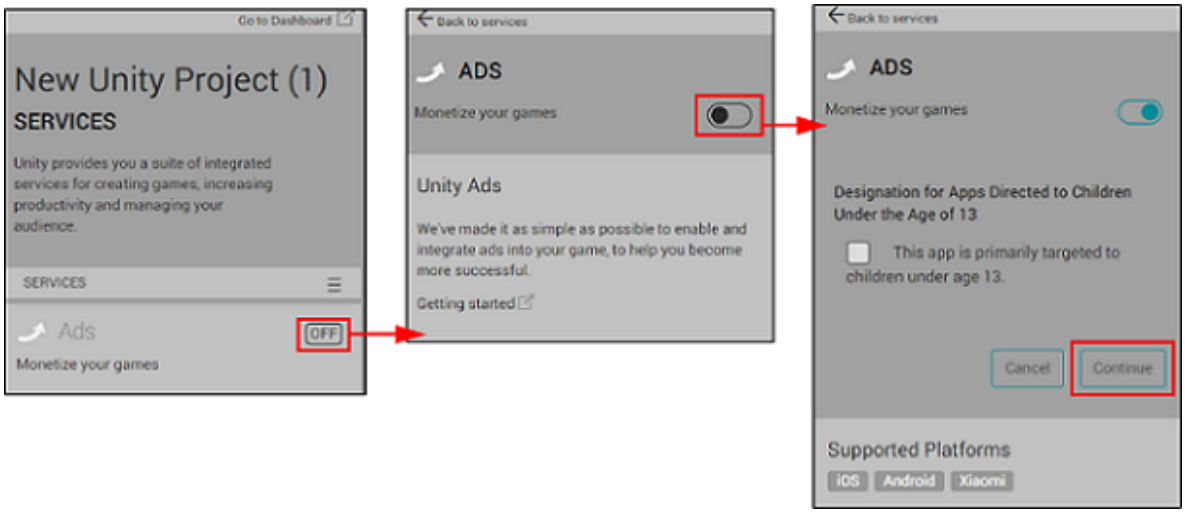
Services Window method

To enable Ads, you need to configure your project for Unity Services. This requires setting an **Organization** and **Project Name**. See documentation on setting up Services.

With Services set up, you can enable Unity Ads:

In the Unity Editor, select **Window** > **General** > **Services** to open the Services window.
Select **Ads** from the Services window menu.
Click the toggle to the right to enable the Ads Service (see image below).

Specify whether your product targets children under 13, by clicking in the checkbox then click **Continue**.



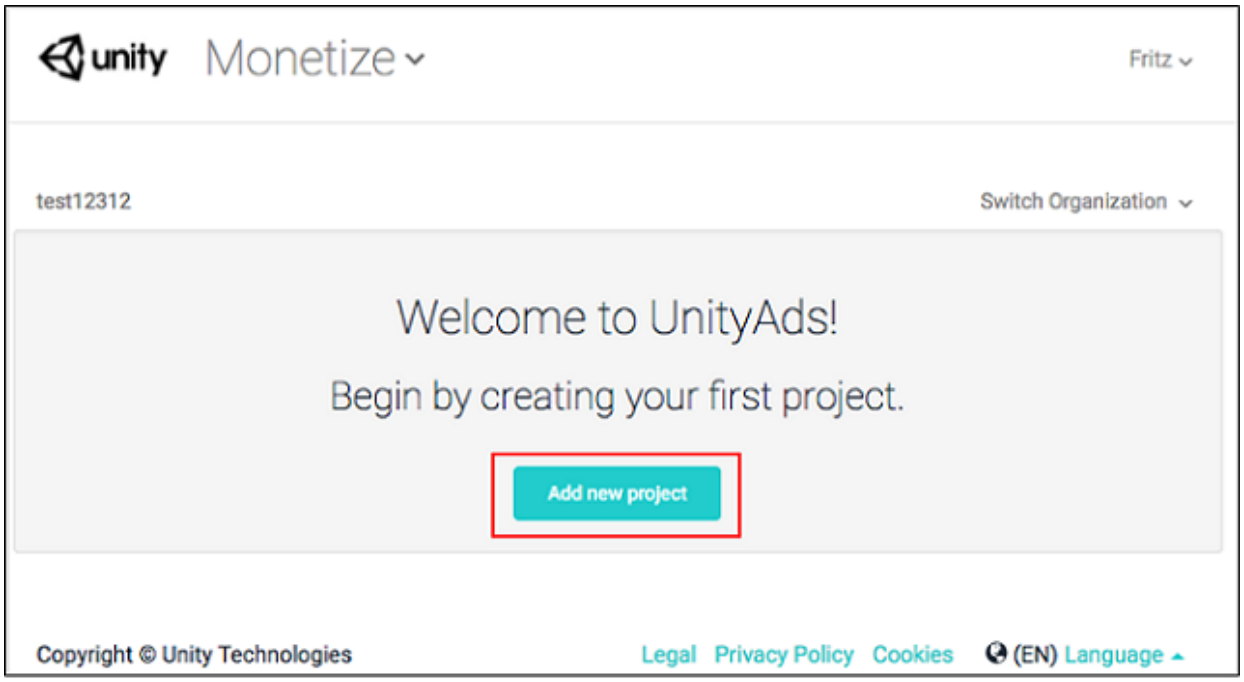
Services window in the Unity Editor (left) and the Ads section of the Services window with the enable toggle (middle, right)

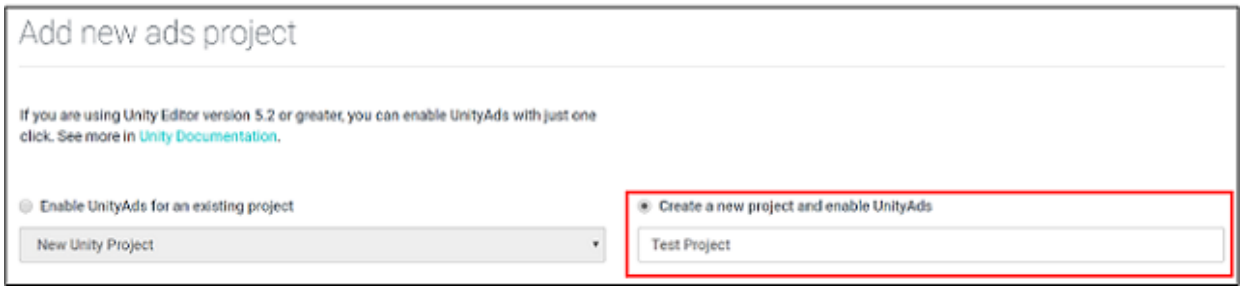
Asset Package method

Before integrating Ads the Asset Package, you need to create a Unity Ads Game ID, as described below.

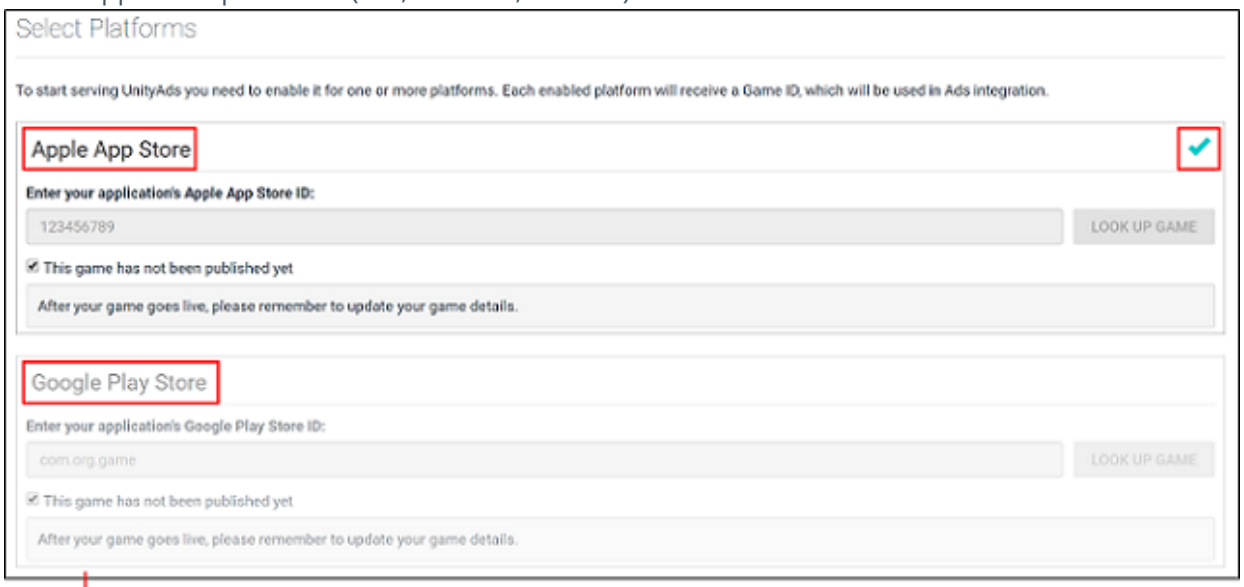
Create a Unity Ads Game ID

In your web browser, navigate to the Unity Ads Dashboard, using your Unity Developer Network UDN Account, and select **Add new project**.





Select applicable platforms (iOS, Android, or both).



Locate the platform-specific `Game ID` and copy it for later.

Integrate Ads to the Asset Package

Declare the Unity Ads namespace, `UnityEngine.Advertisements` in the header of your script (see UnityEngine.Advertisements documentation):

```
using UnityEngine.Advertisements;
```

Inititalize Unity Ads early in the game's runtime lifecycle, preferably at launch, using the copied Game ID string, `gameId`:

```
Advertisement.Initialize(string gameId)
```

Showing ads

With the Service enabled, you can implement the code in any script to display ads.

Declare the Unity Ads namespace, `UnityEngine.Advertisement` in the header of your script (see UnityEngine.Advertisements documentation):

```
using UnityEngine.Advertisements;
```

Call the `Show()` function to display an ad:

```
Advertisement.Show()
```

Rewarding players for watching ads

Rewarding players for watching ads increases user engagement, resulting in higher revenue. For example, games may reward players with in-game currency, consumables, additional lives, or experience-multipliers.

To reward players for completing a video ad, use the `HandleShowResult` callback method in the example below. Be sure to check that the `result` is equal to `ShowResult.Finished`, to verify that the user hasn't skipped the ad.

In your script, add a callback method.
Pass the method as a parameter when when calling `Show()`.
Call `Show()` with the `"rewardedVideo"` placement to make the video unskippable.
Note: See Unity Ads documentation for more detailed information on `placements`.

```
void ShowRewardedVideo ()
{
    var options = new ShowOptions();
    options.resultCallback = HandleShowResult;

    Advertisement.Show("rewardedVideo", options);
}

void HandleShowResult (ShowResult result)
{
    if(result == ShowResult.Finished) {
        Debug.Log("Video completed - Offer a reward to the player");
        // Reward your player here.

    }else if(result == ShowResult.Skipped) {
        Debug.LogWarning("Video was skipped - Do NOT reward the player");

    }else if(result == ShowResult.Failed) {
        Debug.LogError("Video failed to show");
    }
}
```

Example rewarded ads button code

Use the code below to create a rewarded ads button. The ads button displays an ad when pressed as long as ads are available.

Select **Game Object** > **UI** > **Button** to add a button to your Scene.

Select the button you added to your Scene, then add a script component to it using the Inspector. (In the Inspector, select **Add Component** > **New Script** .)

Open the script and add the following code:

Note: The two sections of code that are specific to Asset Package integration are called out in comments.

```
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.Advertisements;

//---------- ONLY NECESSARY FOR ASSET PACKAGE INTEGRATION: ----------//

#if UNITY_IOS
private string gameId = "1486551";
#elif UNITY_ANDROID
private string gameId = "1486550";
#endif

//-------------------------------------------------------------------//

    ColorBlock newColorBlock = new ColorBlock();
    public Color green = new Color(0.1F, 0.8F, 0.1F, 1.0F);

[RequireComponent(typeof(Button))]
public class UnityAdsButton : MonoBehaviour
{
    Button m_Button;

    public string placementId = "rewardedVideo";

    void Start ()
    {
        m_Button = GetComponent<Button>();
        if (m_Button) m_Button.onClick.AddListener(ShowAd);

        if (Advertisement.isSupported) {
            Advertisement.Initialize (gameId, true);
        }

        //---------- ONLY NECESSARY FOR ASSET PACKAGE INTEGRATION: ----------/

        if (Advertisement.isSupported) {
            Advertisement.Initialize (gameId, true);
```

```
            }

            //-----------------------------------------------------------------/

        }

        void Update ()
        {
            if (m_Button) m_Button.interactable = Advertisement.IsReady(placementI
        }

        void ShowAd ()
        {
            var options = new ShowOptions();
            options.resultCallback = HandleShowResult;

            Advertisement.Show(placementId, options);
        }

        void HandleShowResult (ShowResult result)
        {
            if(result == ShowResult.Finished) {
                Debug.Log("Video completed - Offer a reward to the player");

            }else if(result == ShowResult.Skipped) {
                Debug.LogWarning("Video was skipped - Do NOT reward the player");

            }else if(result == ShowResult.Failed) {
                Debug.LogError("Video failed to show");
            }
        }
    }
```

Press **Play** in the Unity Editor to test the Ads button integration.

For further guidance, see the Unity Ads forum.

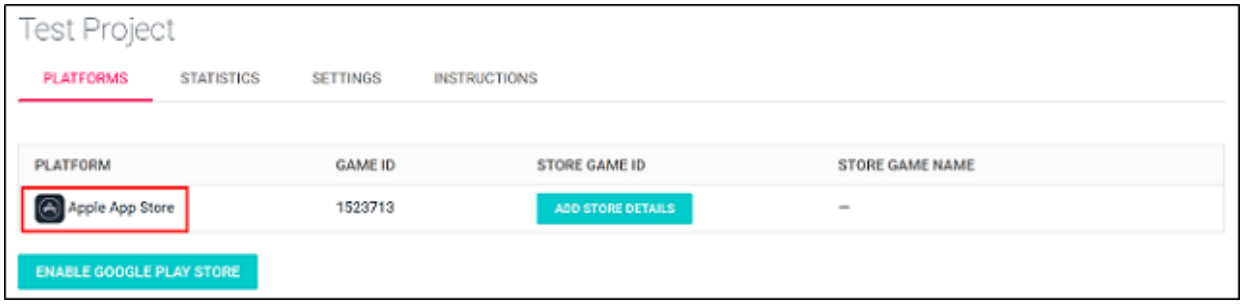
Managing settings in the Ads Dashboard

Use settings to modify placements and other game-specific settings in your project. (See Unity Ads documentation for further information on placements.)

In your web browser, navigate to the Unity Ads Dashboard, using your Unity Developer Network UDN Account, and locate the project for your game.

Unity Ads Dashboard with project selection highlighted

Select an applicable platform (iOS or Android).



Unity Ads Dashboard with platform selection highlighted

Select placement. (See Unity Ads documentation.)

Unity Ads Dashboard showing placement information

2017–08–25 Page published with editorial review

Unity Analytics

Leave feedback

Unity **Analytics** is a simple but powerful data platform that provides **analytics** for your Unity game. Find out who the players are in your game and their in-game behavior.

Important: For information about EU General Data Protection Regulation (GDPR) compliance, see Unity Analytics and the EU General Data Protection Regulation.

Platforms supported:

iOS
Android
Tizen
Universal Windows Platform
Mac, PC, Linux Standalone
WebGL - 5.3 integration and onwards

If you have a question that is not answered in this documentation please ask on Unity Forums.

Legacy notes:

For instructions on integrating analytics into an older project (Pre–5.2), see the Legacy Analytics Integration guide.

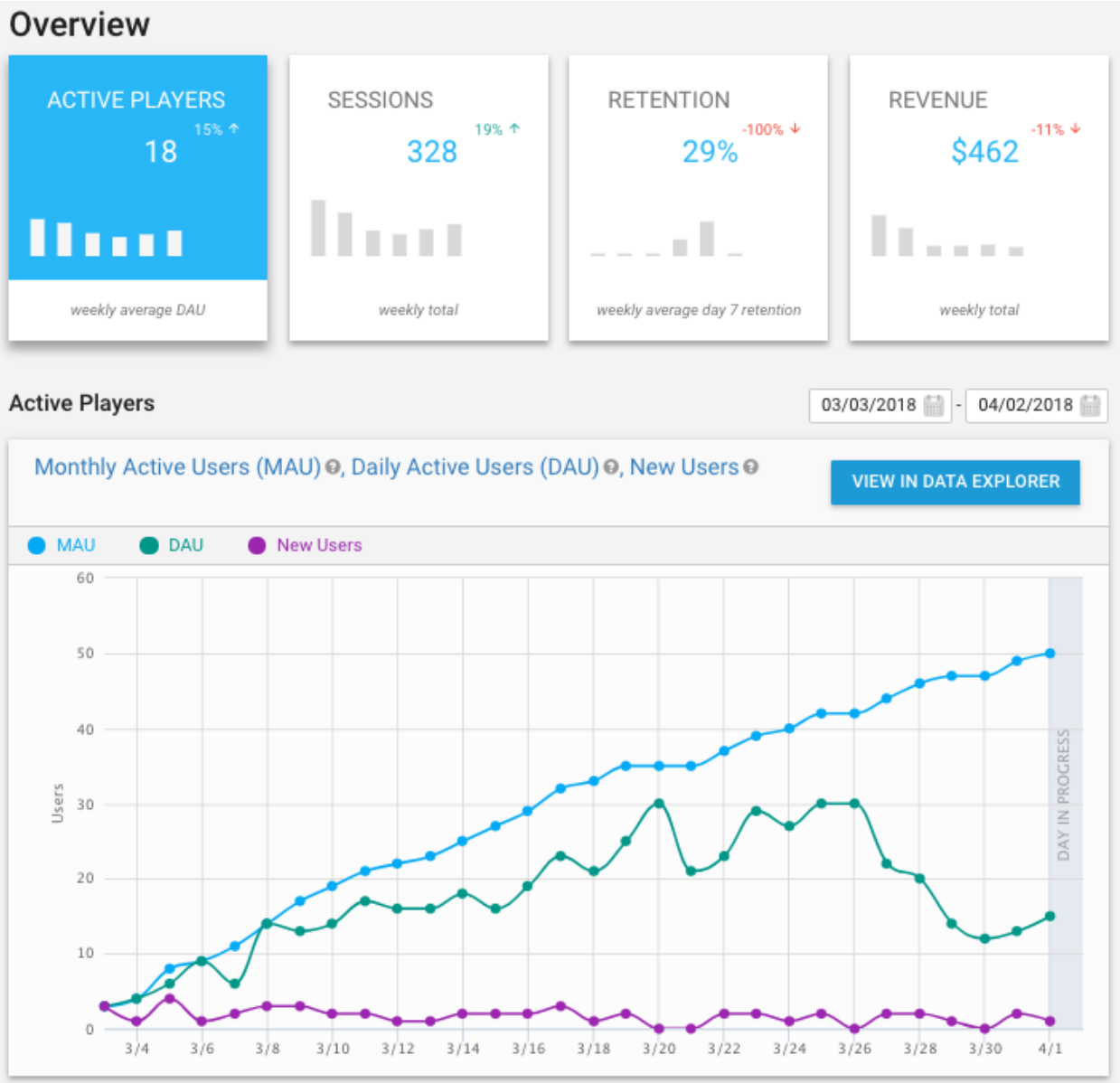
To upgrade an older project which used the Legacy Analytics SDK to the newer built-in Analytics, see the Analytics Upgrade Guides.

• 2017–05–16 Page amended with no editorial review

Unity Analytics Overview

Leave feedback

Unity **Analytics** provides the data you need to manage your relationship with your players.



The Unity Analytics Dashboard

Understand your players; understand why they play your games; understand why they stop playing your games. Make decisions based on data, not guesses.

By simply enabling **Unity Analytics** in your project, you can get data about how many people first start playing your game, how many play every day, how many play in a given month, and other session-based usage data. If you also use the Unity Ads and IAP Services, your analytics data automatically includes the revenue from these sources. This sort of session data paints a valuable picture of the health of your game. It answers question such as: “Am I getting new players?” “How often and how long do they play?” “Do they play again the next day? The next week? The next month?” “How much revenue am I making per player on a given day?”

By dispatching Standard Analytics Events when players perform key actions in your game, you can extend the analytics data available so that it reveals what your players are actually doing when they play. Do they make it through your onboarding tutorial? Do they visit your IAP store? Do they progress through your game levels as you expect?

Analytics data can help you focus on the most fruitful areas to improve your application to better serve your players. You can make these changes by releasing updates, but you can also create **Remote Settings** that allow you to adapt and tune your game

without an update.

By understanding your players and how they play your games, you can make your games better. And when the data shows that your players are changing, you can react and adapt.

See Analytics Metrics, Segments, and Terminology for definitions of specific terms used throughout the Analytics documentation.

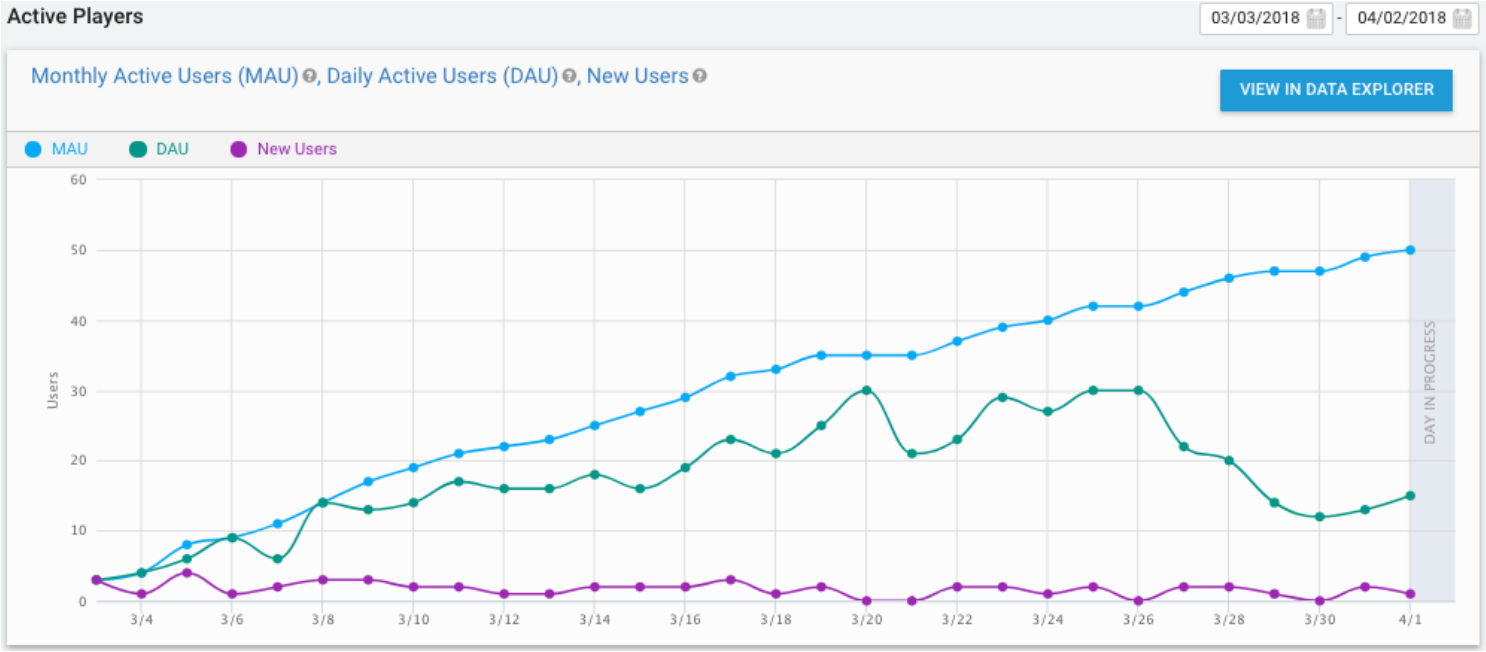
Core usage data

When you enable Analytics, Unity tracks core usage metrics on supported platforms without any effort or implementation on your part. These core metrics include:

New installs
Daily active users (DAU)
Monthly active users (MAU)
Total sessions
Sessions per user
Time spent in app
User Segments for Country and Platform

These are all good metrics to use to monitor the health of your game. They tell you how many people are installing your game, how often they play, and how long they stick with it. Many external factors also influence these numbers, but you should watch them for signs of problems that you can correct.

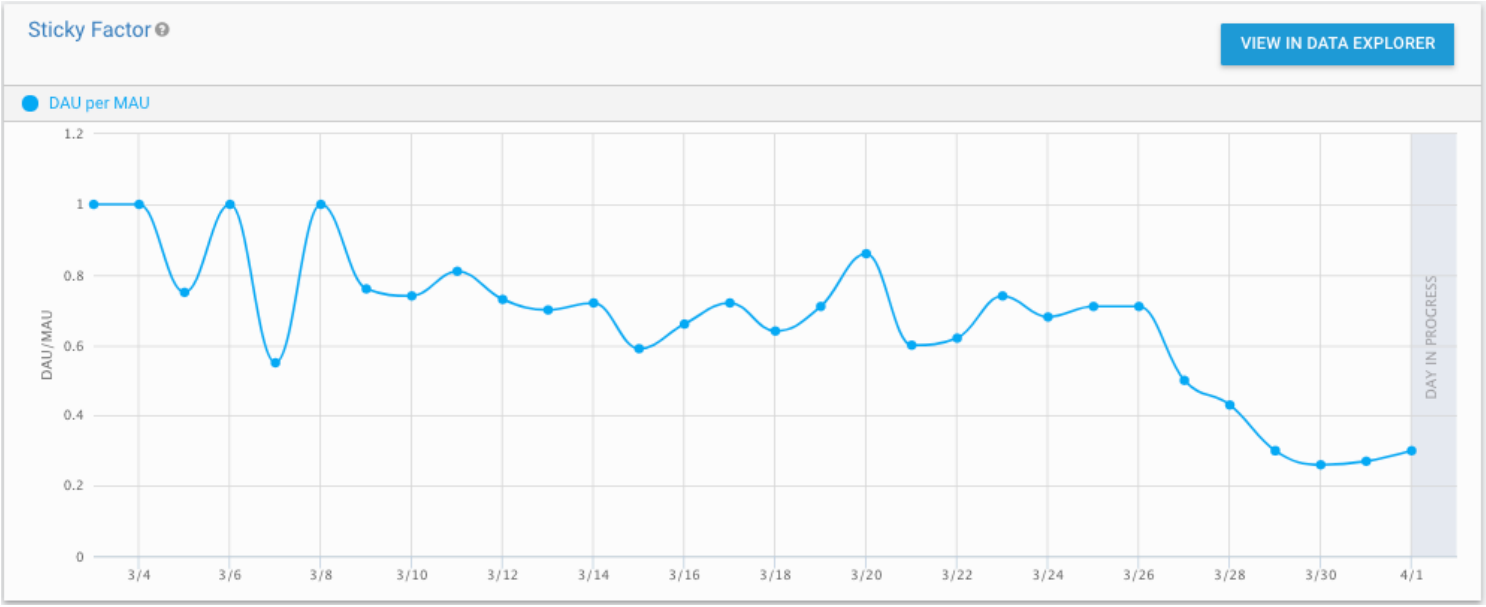
The following chart, displayed on the Overview tab of your Analytics Dashboard shows three data plots: **New Users**, **DAU** (Daily Active Users), and **MAU** (Monthly Active Users):



DAU, MAU, New Users chart

The **New Users** plot shows how many people play your game for the first time on a given day. Since new players won't have had much, if any, exposure to your game, this chart generally reflects external factors. If you launch a marketing campaign or other promotion to attract new players, you would hope to see a bump in the **New Users** plot.

The **DAU** plot shows how many of your players played on a given day. Similar to the **DAU** plot, the **MAU** plot shows how many of your players played in a given 30-day period.



Sticky Factor chart

The **Sticky Factor** chart plots the percentage of your monthly players who played on a given day (DAU/MAU). The general idea behind this metric is that it shows how often your players return to play day-after-day. (In other words, how likely players are to "stick" with your game.)

High active player numbers can indicate popularity, but can be tricky to interpret in isolation. In particular, these metrics don't separate the influence of new versus returning users. For example, an influx of new players could mask the loss of more established players. (The loss of players is called "churn" in common analytics jargon.) **DAU**, **MAU**, and **Sticky Factor** metrics are often used in the Games industry because DAU and MAU have historically been the most widespread publically-available data with which to compare different games.

See Analytics Dashboard for information on how to view your analytics data.

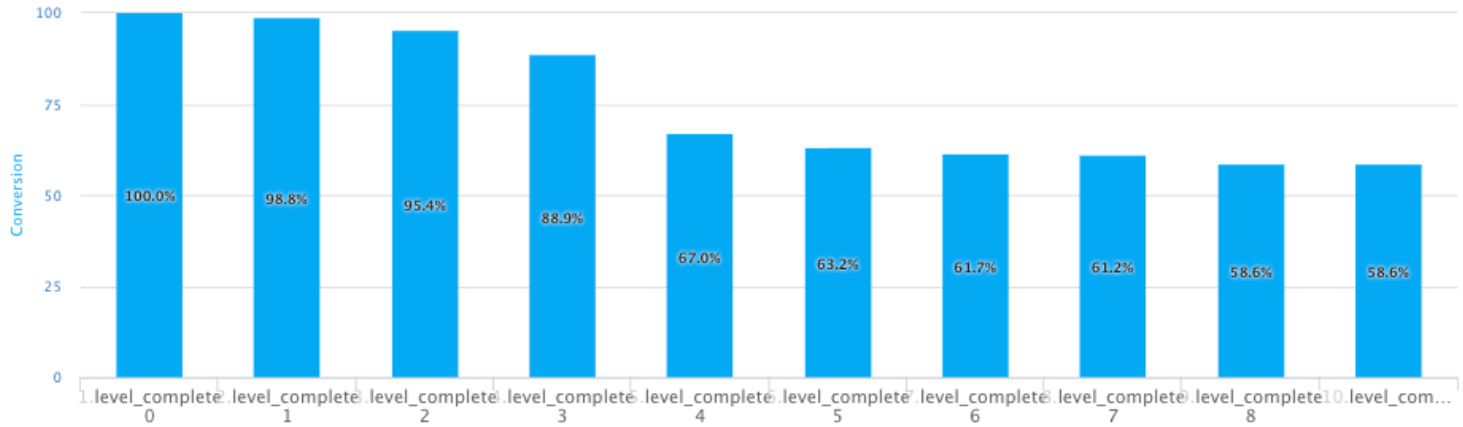
Player behavior data

To learn about game-specific player behavior, you can send analytics events from your game at the appropriate times. Unity provides easy-to-use APIs for sending both standardized and fully-customized events. You can use these events to instrument your games to monitor player behavior, particularly in the following areas:

Onboarding — are players making it through your onboarding mechanics such as tutorials or starting levels?
Progression — are players progressing through your levels?
Economy — are your game economies working out as expected?
Design validation — are your game design choices working out as you thought they would?
Application validation — are all areas of your application being utilized as you expect? Are there parts that players ignore or don't notice?
Monetization — are your monetization strategies optimal? Are there impediments to players carrying out purchases?

See Analytics Events for information on gathering player behavior data.

To help you analyze player behavior, the Analytics Dashboard includes a funnel builder. A funnel is a display that shows how your players progress through a linear sequence of steps. For example, you could create a funnel for a tutorial that showed what percentage of your users made it through the tutorial steps. Funnels are useful for identifying places in your application where you lose players.



Level progression funnel

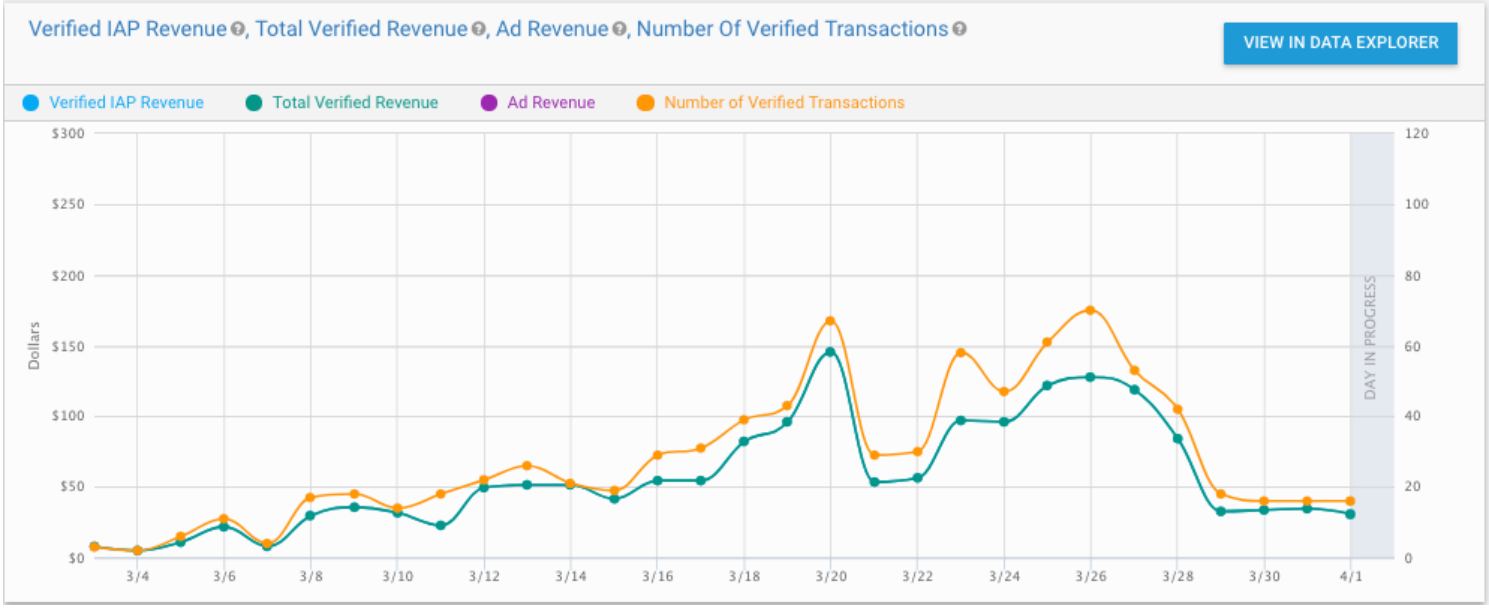
This example funnel shows player progression through a hypothetical game. Each step in the funnel represents the completion of a game level. While you can always expect some drop off from level to level, too large a drop after a given level could indicate a problem on that level. The funnel won't tell you the cause – it could be a game play problem, a bug, or the level might be too difficult – but the funnel does indicate a worthwhile area for investigation.

See Funnels for information on implementing funnels.

Revenue data

You can analyze in-app purchase (IAP) revenue along with your other analytics data. If you use the **Unity IAP** Service, the revenue data is available automatically. Otherwise, you can send an event to the Analytics Service whenever a player makes a purchase. If you use the Unity Ads Service, you can also analyze the resulting revenue on the Analytics Dashboard. (There isn't any way to report outside ad revenue to the Analytics Service, though.)

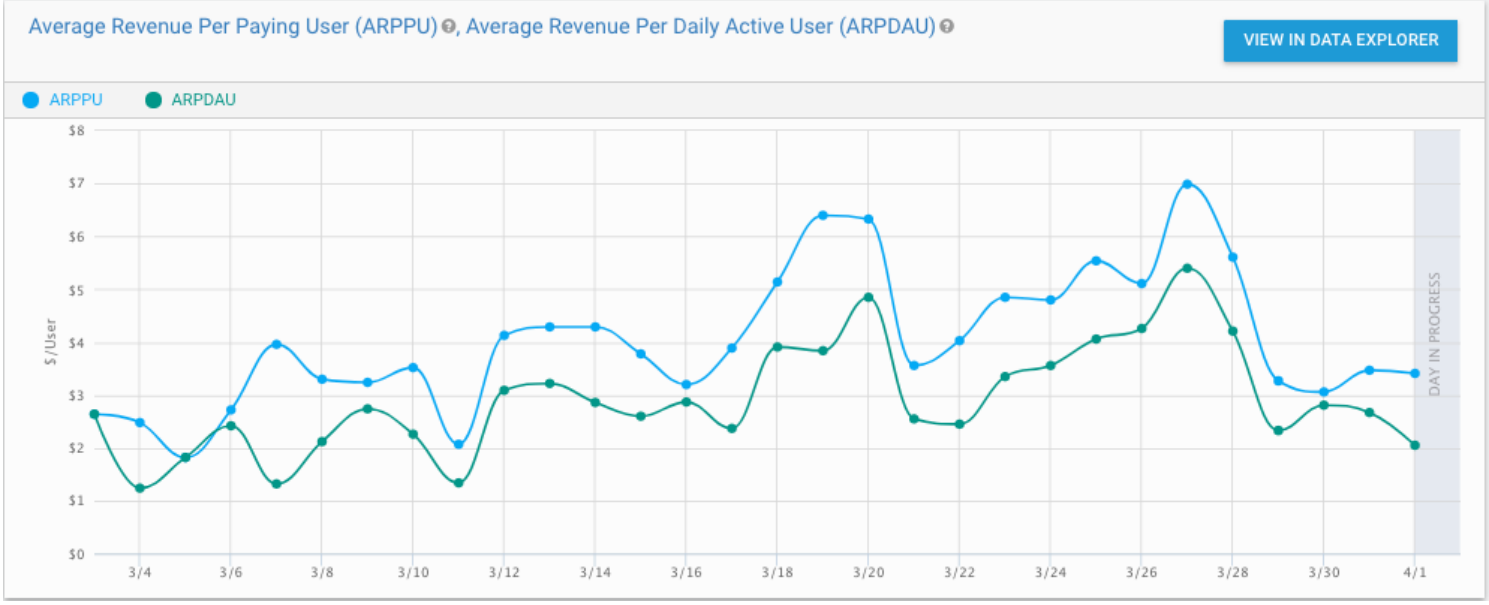
The following charts, displayed on the Overview tab of your Analytics Dashboard, show the daily revenue numbers for your project:



Revenue totals

The first chart shows verified in-app purchase revenue and transactions and advertising revenue from the Unity Ads Service. The charts on the **Overview** tab only show **verified IAP revenue**, but you can view all reported revenue in the **Data Explorer**. Unverified revenue sources include test transactions, fraudulent transactions, transactions from platforms that do not support receipt verification (like Amazon and the Windows store), and transactions reported using missing or incorrect information or without configuring the required store API keys. When you use the Unity IAP Service, IAP transactions are automatically reported and verified. When you use an external IAP API, you can report and verify IAP revenue through the Unity Analytics API.

The second chart shows the daily averages of revenue per paying user and per active user:



Average Revenue

Average Revenue Per Paying User (ARPPU) shows the average amount spent by those players who make IAP transactions on a given day. The **Average Revenue Per Daily Active User** (ARPDAU) shows the average revenue from all users and includes both IAP revenue and advertising revenue on a given day.

The overview charts show revenue for all users. To better understand how different groups of players monetize, use the Data Explorer to view revenue by segment. The standard segments include lifecycle cohort, geography, monetization category, demographics (when reported), and platform. You can also define your own segments.

The Data Explorer page of the Dashboard provides several additional monetization metrics and segments:

Monetization Metrics:

ARPDAU (Average Revenue Per Daily Active User)
ARPPU (Average revenue Per Paying User)
Number of Unverified Transactions
Number of Verified Transactions
Total IAP Revenue
Total Verified Revenue
Unverified IAP Revenue
Verified IAP Revenue
Verified Paying Users

Ad Metrics:

Ad ARPU (Average Revenue Per User)
Ad Revenue
Ad Starts
Ads per DAU
eCPM (estimated Cost Per Thousand Impressions)

Monetization Segments:

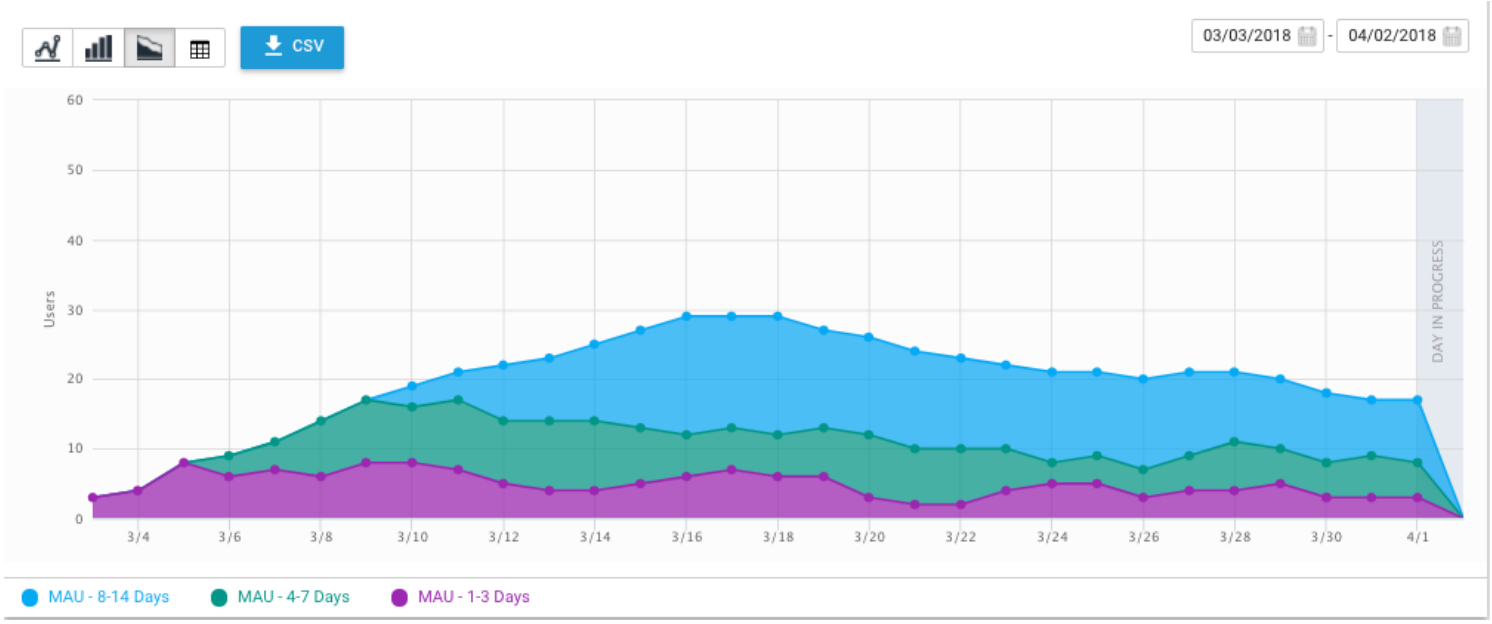
Whales
Dolphins
Minnows
Never Monetized
All Spenders

See Monetization for more information about revenue analytics.

Data segmentation

Segments help you better understand player behavior. The standard, predefined segments group players by the following categories:

Lifecycle – group players by the number of days since install. Lifecycle cohorts are useful for analyzing how player behavior changes with more experience in your application.
Geography – group players by country.
Monetization – group players by spending category. Monetization categories help you understand the behavior of players by how much revenue they generate.
Platform – group by the player's computer OS or device type.



The MAU metric segmented by lifecycle cohorts

You can use the **Segment Builder** to define your own categories. However, you can only use any new categories that you make on data collected after those categories are created. For example, if you add a segment for players from Gibraltar, the segment is initially empty; only new activity from players in Gibraltar will show up in the segment.

You can use segments in the Data Explorer and Funnel reports.

For more information about creating your own segments, see Segment Builder.

Remote Settings

The **Remote Settings** feature allows you to change the values of variables in your games from your Analytics Dashboard. You can create any number of **Remote Settings** while developing your game and assign them to **GameObject** variables in the Editor. After launching your game, you can change the values of these variables at any time, allowing you to adapt game behavior without releasing an update.

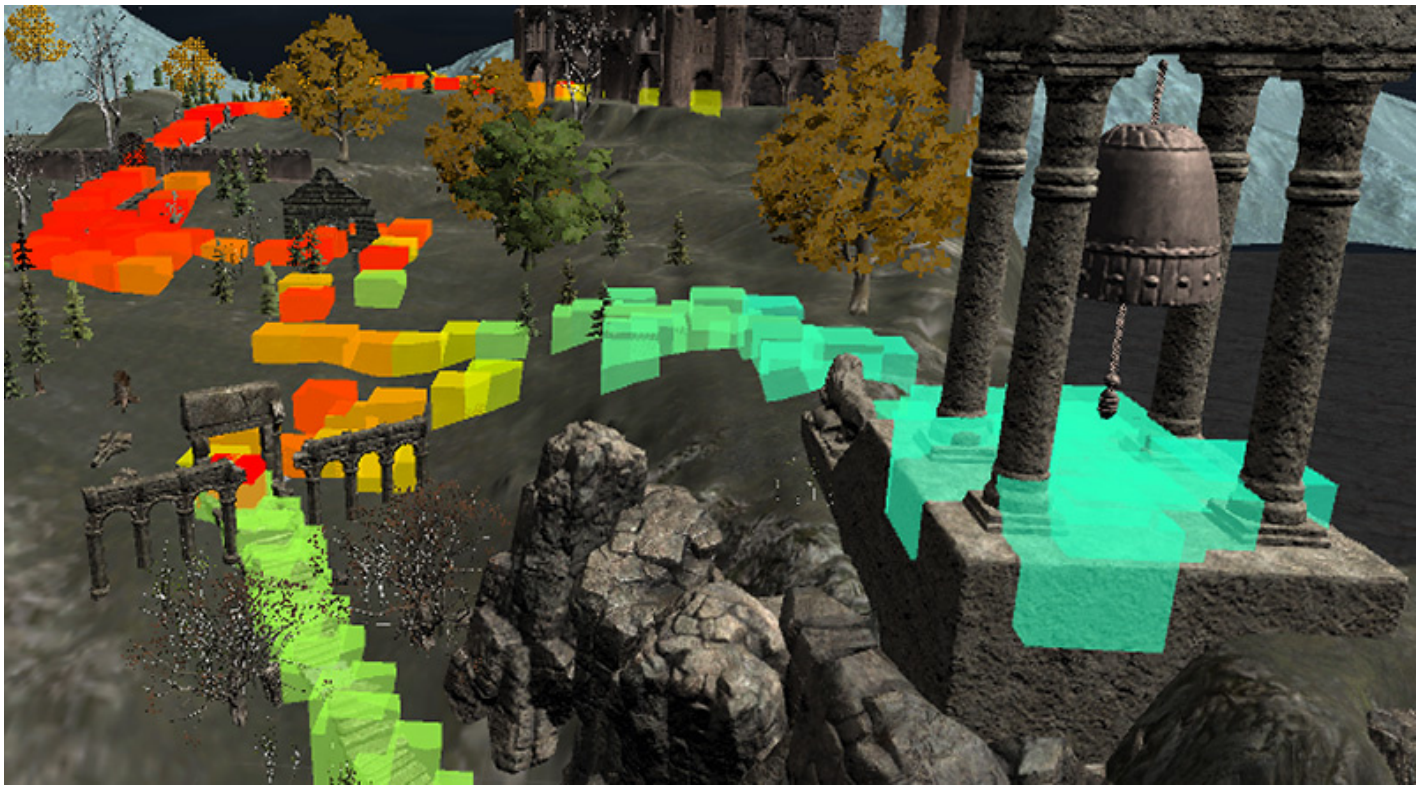
You can use **Remote Settings** for tasks such as:

Adjusting game-play tuning variables
Enabling or disabling features
Changing graphical themes

See Remote Settings for more information.

Heatmaps

Heatmaps visualize Analytics events spatially in your game.



A heatmap displays Analytics events in the Unity Editor where they occurred in a scene

Use heatmaps while playtesting your game to identify performance and gameplay bottlenecks. Because they require a high volume of Analytics events, using heatmaps after your game has shipped and is in production is not supported.

See BitBucket documentation on Heatmaps for more information.

Heatmaps require **Raw Data Export**, which is only available with a Unity Pro subscription.

Accessing your raw data

When the reports available on the Analytics Dashboard don't support the type of analysis you want to perform, you can export your analytics events using **Raw Data Export**. You can export data from the Analytics Dashboard or using the Raw Data Export REST API in either JSON or tab-separated-value format. From there, you can import the data into a database or analysis tool.

Raw Data Export requires a Unity-Pro subscription. See Raw Data Export for more information.

Supported platforms

The Unity Analytics Service supports the following platforms:

iOS
Android
Tizen
Windows Phone 8.1
Windows Store 8.1 (Desktop)
Windows Store 10.0 (Desktop)
Mac, PC, Linux Standalone
WebGL - 5.3 integration and onwards

2018–06–22 Page published with editorial review

New feature in Unity 2017.1

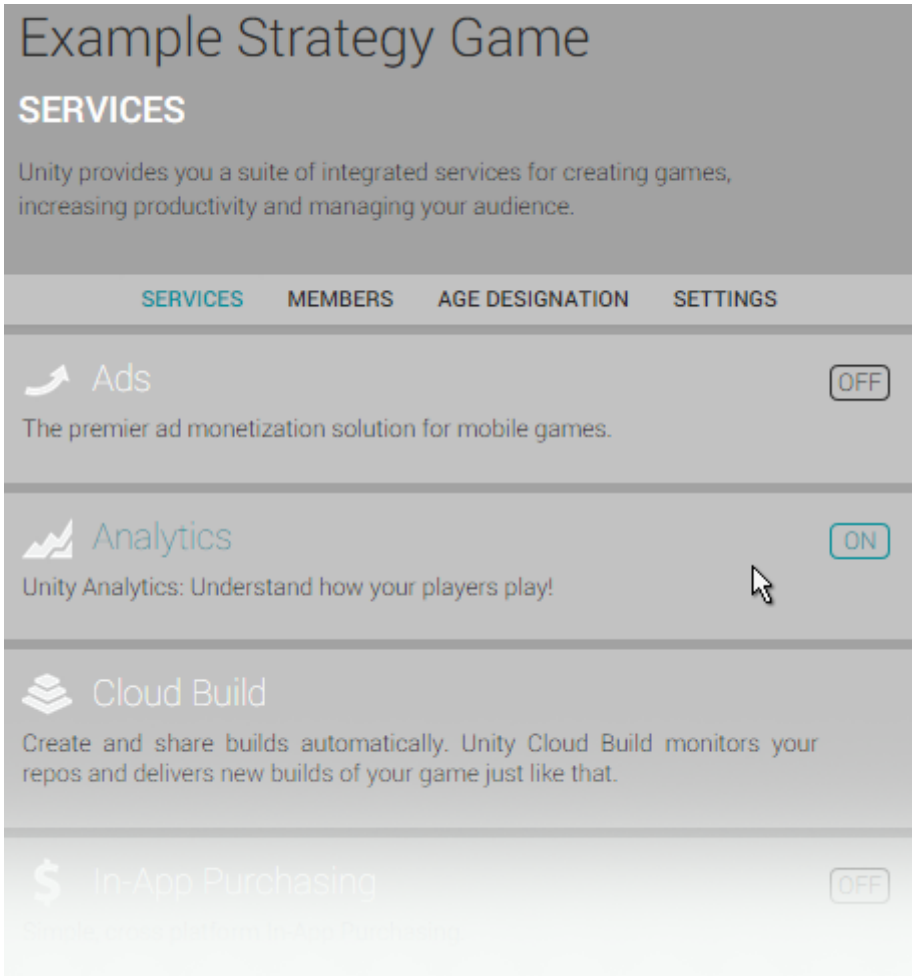
Removed Demographics section.

Setting Up Analytics

Leave feedback

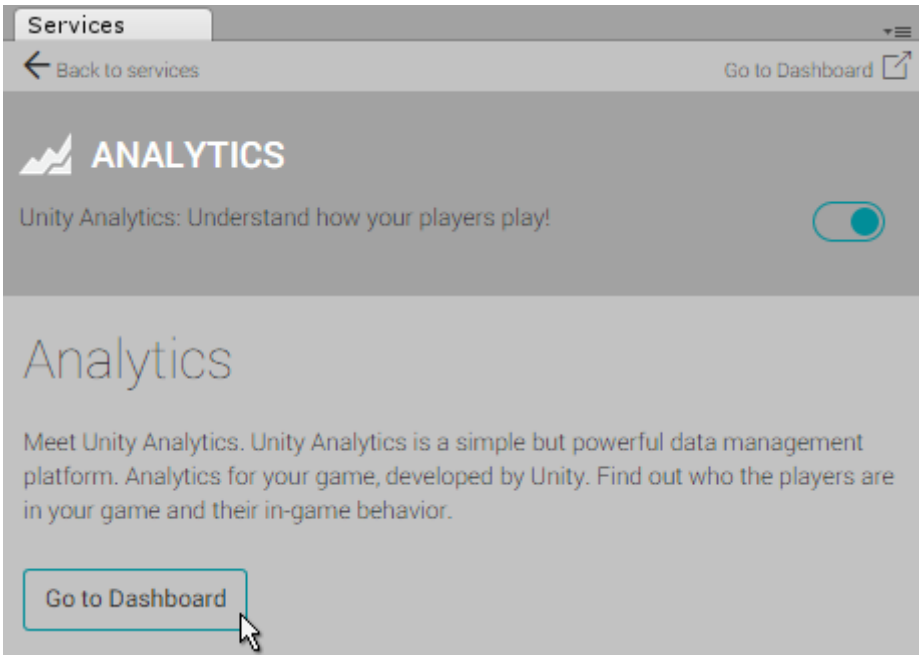
Once you have set up your project for Unity Services, you can enable the Analytics service.

To do this, in the Services window, select **Analytics** and then click the **OFF** button to toggle it **ON**. If you haven't already done so, at this stage you will need to complete the mandatory **Age Designation** field for your project. (Again, you may have already done this for a different Unity Service, such as Ads). This age designation selection will appear in the Services window.



Enabling Analytics within your project

You must then hit **Play** in your project to validate the connection to **Unity Analytics**. The Unity Editor acts as a test environment to validate your Analytics integration. With Analytics enabled, when you press the **Play** button, the Editor sends data (an "App Start" event) to the analytics service. This means you can test your analytics without having to build and publish your game. Once you have pressed play, you can check that your project was validated by going to the Analytics Dashboard for this project. To get there, in the services window, click Services -> Analytics -> Go To Dashboard.



The **Go to Dashboard** button in the Analytics section of the Services window opens the dashboard in a web browser. The dashboard contains tools for visualizing, analyzing, and taking action based on your Analytics data. See Analytics Dashboard.

2017–08–29 Page published with editorial review
New feature in Unity 2017.1

Analytics Dashboard

Leave feedback

The Unity **Analytics** Service provides a web dashboard for:

Visualizing your Analytics data
Performing data segmentation and analysis
Conducting live operations
Configuring **Unity Analytics** services
The Analytics dashboard contains several pages for these purposes:

Analytics section:

Overview – A pre-built visualization of common key performance indicators derived from your Analytics data.
Data Explorer – Build, view, and export reports of your Analytics data.
Funnel Analyzer – Analyze player progress and behavior in your game by building funnels.
Segment Builder – Define customized segments to characterize your player population.
Livestream – View **Analytics events** as they occur. (Plus and Pro subscriptions only)
Raw Data Export – Export your unprocessed event data. (Pro subscriptions only)
Event Manager – List and manage your Custom Events.
Market Insights – Consult the Market Insights page to research trends in the computer hardware used to play Unity games.
Optimization section:

Tutorial Manager – (Closed Beta) Use machine learning to improve onbaording and retention. Also provides a toolkit to make tutorial development easier.
LiveTune – (Closed Beta) optimizes content quality for every device in real-time to deliver the best app experience and increase user engagement and player retention. See LiveTune Manual.
A/B Testing – (Open Beta) Conduct A/B testing based on Remote Settings.
Remote Settings – Change application variables remotely and without an application update. Find the Remote Settings page in the **Optimization** section of the dashboard.
Settings section:

Configure – Project Analytics settings. Find the Configure page under **Settings** > **Analytics Settings** on the dashboard.

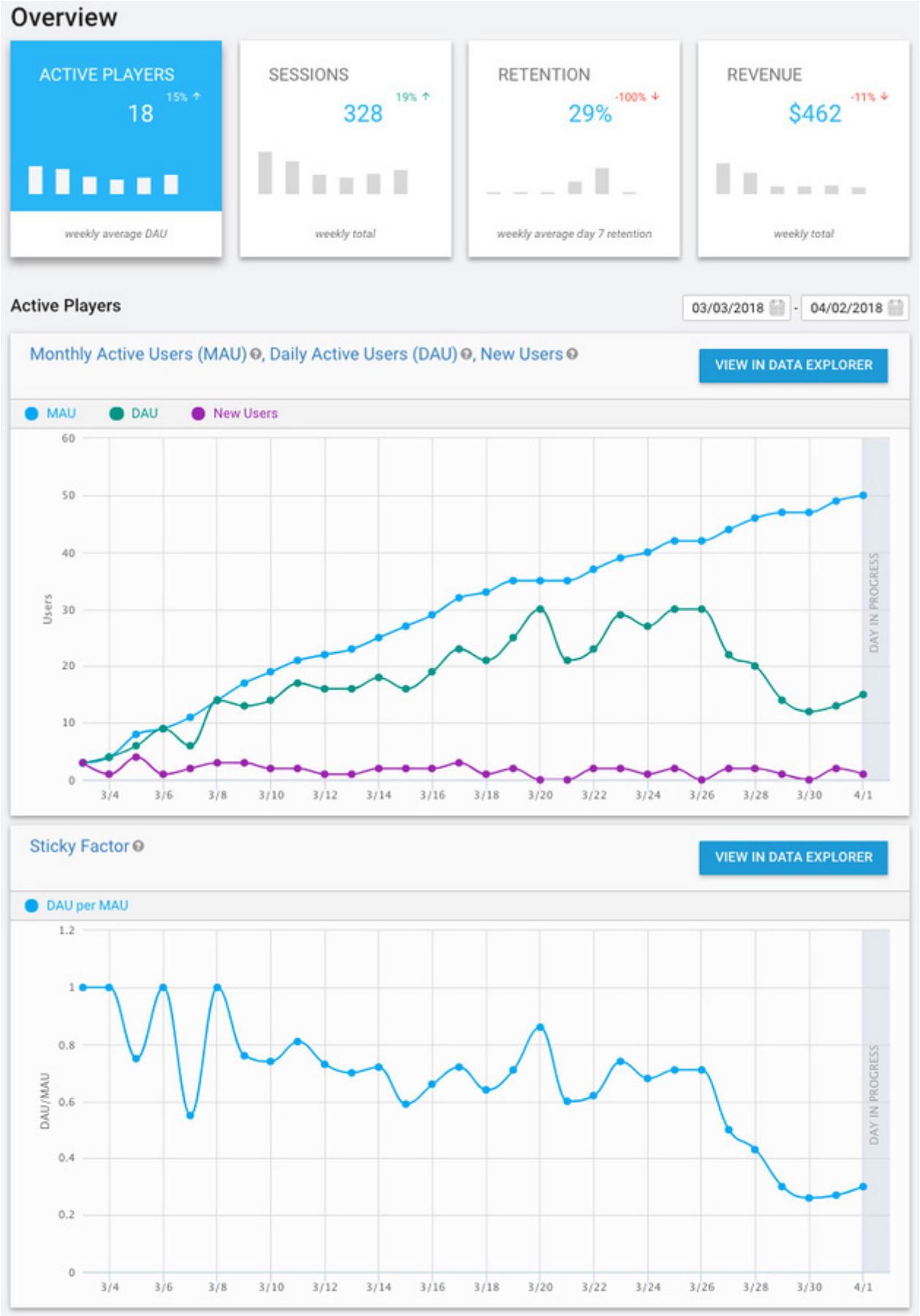
2018–08–23 Page published with editorial review
New feature in Unity 2017.1

Overview page

Leave feedback

The Overview page of your Unity **Analytics** Dashboard provides a number of predefined charts that track commonly used metrics and key performance indicators. The overview page presents the charts in the following sections:

Active Players	Shows how many players start to play your game as well as return to it over time.
Sessions	Shows how often and how long the average player plays your game each day.
Retention	Shows the percentage of players who return to the game at significant intervals.
Revenue	Shows reported, verified In-App Purchase (IAP) and Unity Ads revenue.



The Unity Analytics Overview tab showing the Active Player charts

For each chart displayed in these sections, you can select the date range to display. You can also click the **View in Data Explorer** button to view the report in the DataExplorer tab where you have many more options for

selecting, formatting, and analyzing report data.

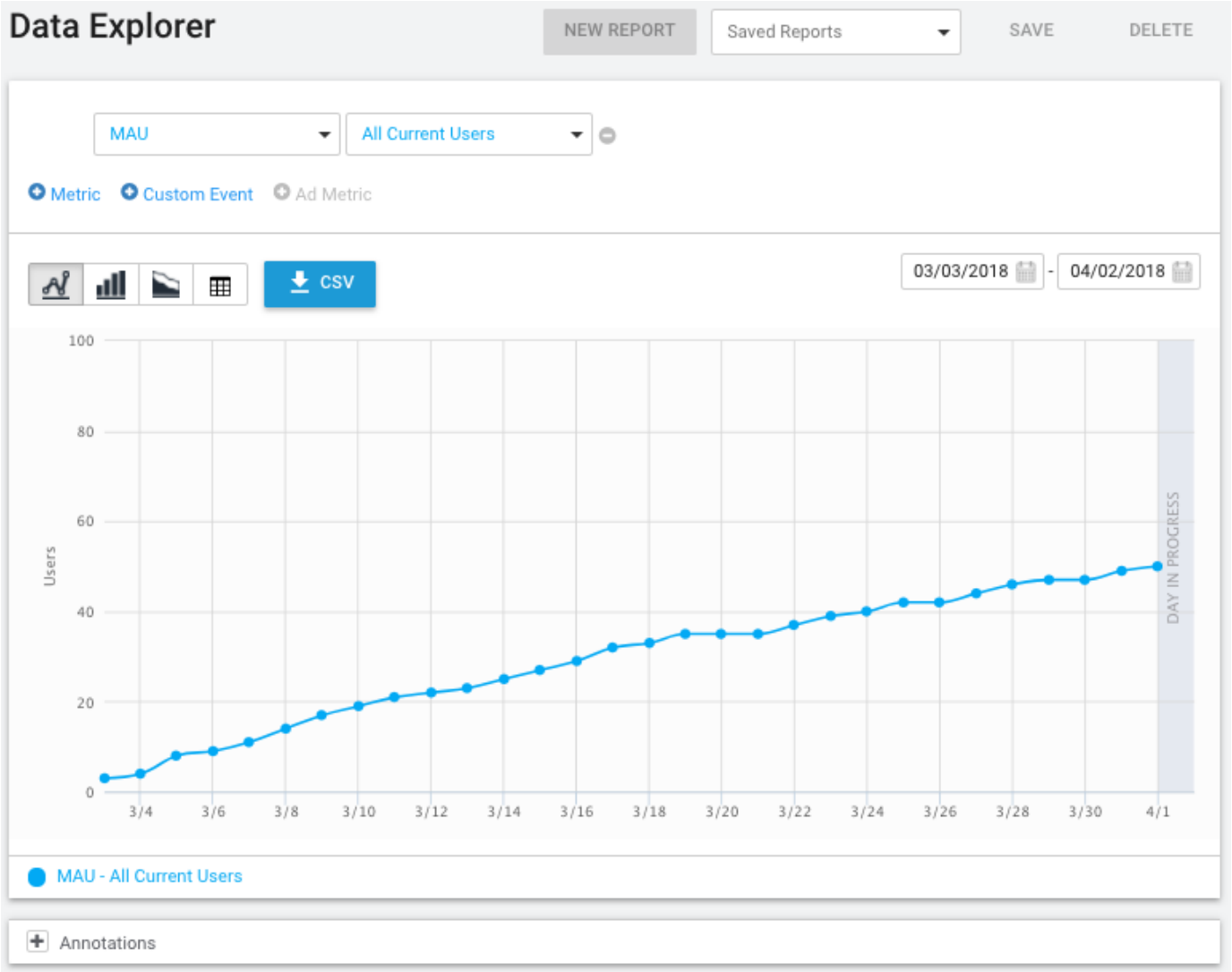
For more information about the metrics used on the Overview page and elsewhere on the Analytics Dashboard, see Analytics Metrics, Segments, and Terminology.

2017–08–29 Page published with editorial review
New feature in Unity 2017.1

Data Explorer

Leave feedback

Build reports for your Unity **Analytics** metrics and events using the **Data Explorer**. The **Data Explorer** reports show how selected metrics and custom events change over time.



A report in the Data Explorer page showing MAU over time

Create a new report

To create a Data Explorer report:

Navigate to the Analytics dashboard for your project.
Click **Data Explorer**.
Add **Metrics**, **Custom Events**, and **Ad Metrics** to the report by clicking the + buttons next to the desired type of data. (The Ad Metrics option is disabled for projects not using Unity Ads.)

Set the date range and chart type as desired. See Report Options for more information.
Note: The default report includes a single metric. You can change the metric type and segment for this item. (To delete this item, you must first add a second item – you cannot delete the last data item on a report.)

To save a report for future viewing:

Click the **New Report** button.

On the dialog that opens, enter a name for the report and click **OK**.

To view a saved report in the future, click the **Saved Reports** dropdown and select the report name in the list. You can adjust the date range to view the same report with your latest data.

Copy a report

To copy an existing Data Explorer report:

Navigate to your Analytics dashboard for your project.

Click **Data Explorer**.

Click the **Saved Reports** dropdown and select the report name in the list.

Adjust the date range, chart type or other options as desired. See Report Options for more information.

Click the **New Report** button.

On the dialog that opens, enter a name for the report copy and click **Ok**.

Download report data

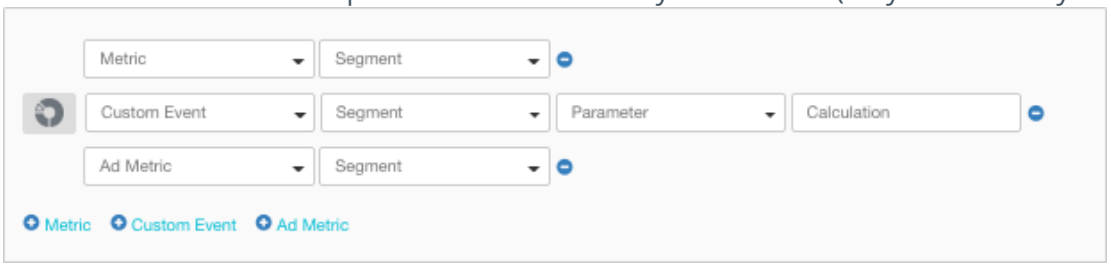
To download a comma-separated value file (CSV) containing the data in a report:

Navigate to your Analytics dashboard for your project.
Click **Data Explorer**.
Load a saved report or create a new one.
Adjust the date range or other options as desired. See Report Options for more information. Click the **CSV** button to download the data for the currently displayed report.
Note: CSV download exports precisely what you see in the chart/table. If you have a Unity Pro subscription, you can also download all of your Analytics data in raw form from the Raw Data Export page or through the Raw Data Export REST API.

Report options

You can add the following types of data to a report:

Metric – Analytics metrics are computed from the Core **Analytics events** sent automatically whenever someone plays your game. These metrics include such things as how many people played your game, how long they played, and how much money they spent on in-app purchases.
Custom Event – Custom events are the events that you dispatch from your game to measure behavior or other factors unique to your game. See Custom events for more information about defining and using custom events.
Ad Metric – Ad metrics report data from the Unity Ads Service (only available if you are using Unity Ads.)

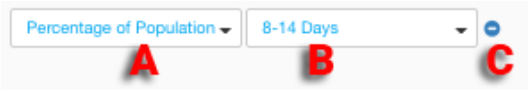


You can add up to twenty data items to a single report. The charts will display the appropriate units for the vertical scale as long as there are no more than two data items that use different units of measure.

You can set the date range for a report and select different chart types.

Metric and Ad Metric options

When you add a **Metric** or **Ad Metric** to the chart, you must set the name and can choose a data **Segment**:



A metric data item on a report
A – The metric name
B – The segment of the data to show
C – Delete the metric from the report

For each data item, you can specify a **Segment** of the data to display. A **Segment** is a subset of the data for that item, selected according to a set of rules. For example, the **Android Users** segment includes data only for players on the Android platform, while the **iOS Users** segment includes data only for players on the **iOS** platform.

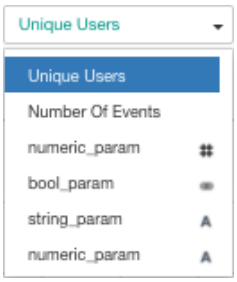
The Analytics Service includes a set of standard segments. You can also create your own custom **Segments**. Use the Segment Builder to view the definitions of the standard segments and to build your own custom **Segments**. Note that when you create a custom **Segment**, it only applies to newly collected metrics and events. Older data is not reprocessed for inclusion in a new **Segment**.

Custom event options

When you add a **Custom Event** to a report, you can set additional options based on the event parameters.

A custom event entry on a report has two extra fields (and a pie chart button)
A – Show a pie chart (only for categorizable data items)
B – The name of the **Custom Event**
C – The **Segment** of the data to show
D – The custom event **Parameter** name and display options
E – How to aggregate parameter values
F – Delete the custom event from the report

When a **Custom Event** has parameters, you can choose how the data for that event is displayed on the chart. You can show the count of each distinct value received or, for numeric properties, the sum or average of the values of all the events received:



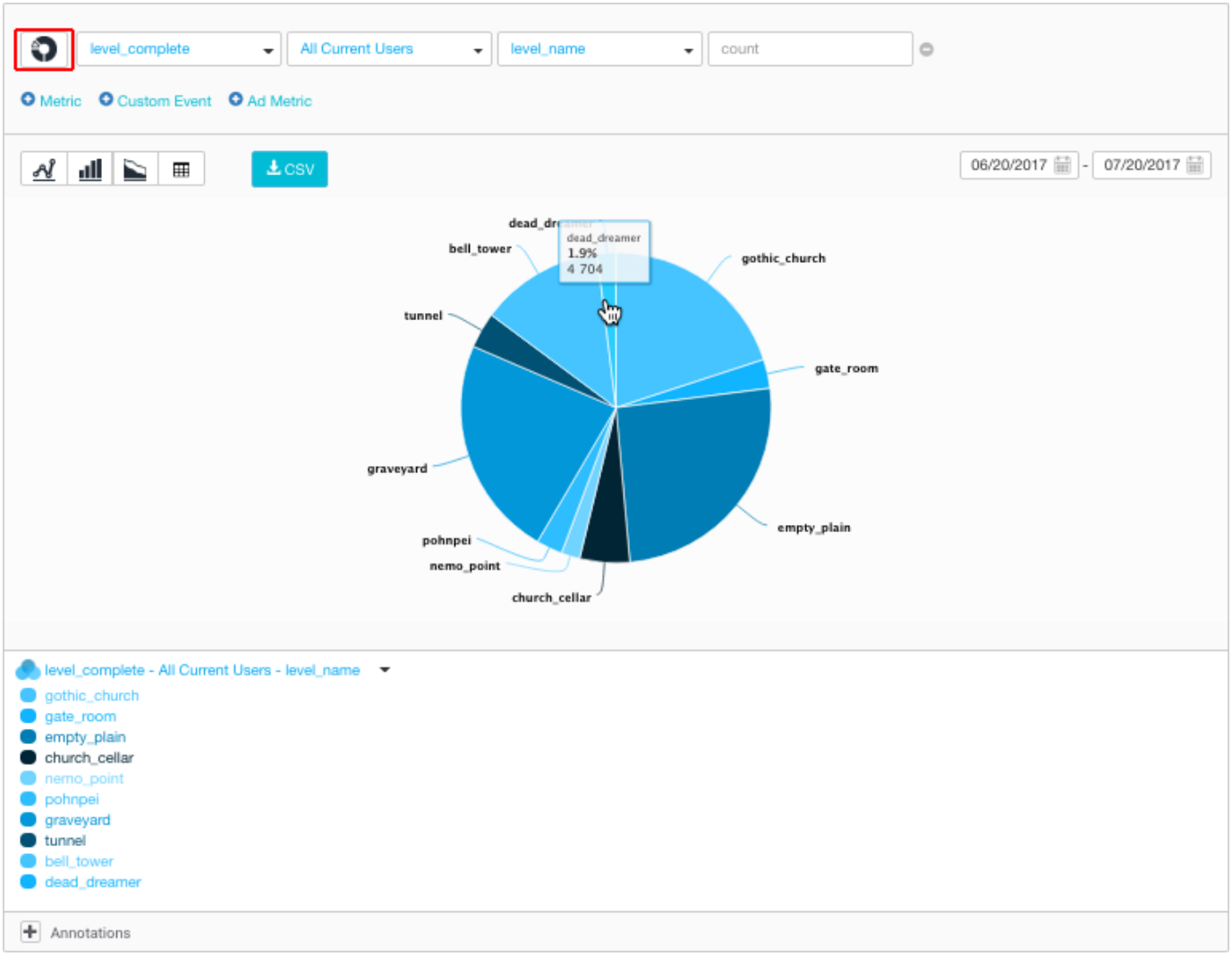
Parameter display options

The symbols displayed after the **Parameter** name indicate how the **Parameter** is displayed. The string (A) and boolean (Venn diagram) symbols indicate that the **Parameter** values are categorizable into distinct groups. The numeric symbol (#) indicates that the **Parameter** value is a number that can be averaged or summed in addition to being counted. Note that numeric **Parameters** appear in the list twice, once with a string symbol and once with a numeric symbol, giving you the option of treating a numeric **Parameter** as either a categorizable or a aggregatable value.

When showing categorizable **Parameter** values, only the ten most common values are broken out into individual groups. The rest of the values are lumped together in an **Other** group. Thus, treating numeric **Parameters** as categorizable strings is generally only useful for **Parameters** with a limited set of distinct values.

You can view the set of distinct values at the bottom of the chart. You can also view a pie-chart of the values by clicking the pie-chart button in front of the data item. Only a single pie chart can be displayed at a time.

Note: To download the data in a categorizable-data pie chart, click the pie-chart button next to the Custom Event before clicking the **CSV** button.



A pie chart for a level_complete event

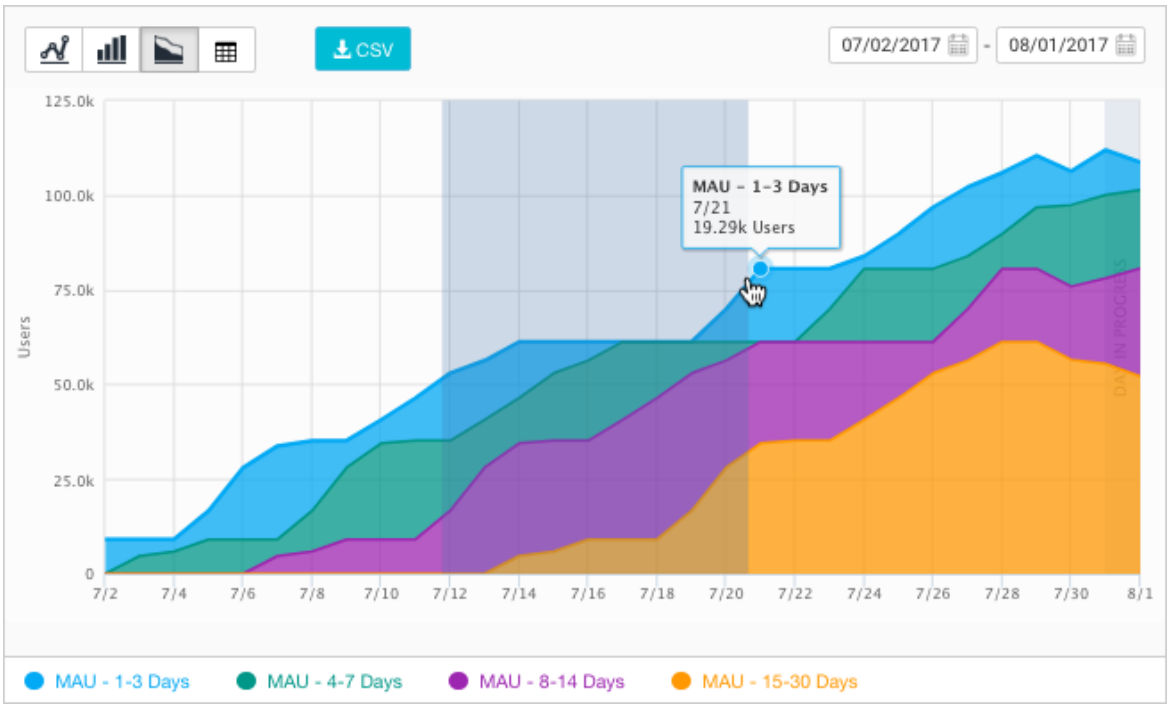
Chart types

You can choose the following chart types for a report:

Spline – A smoothed line chart
Column – A bar chart
Stacked – A stacked area chart
Table – A table of values
Pie – A pie chart (only available for individual, categorizable Custom Event parameters)

Chart Zoom

You can zoom in on the **Spline**, **Column** and **Stacked** charts by clicking on the chart and dragging across the section you want to view in more detail.

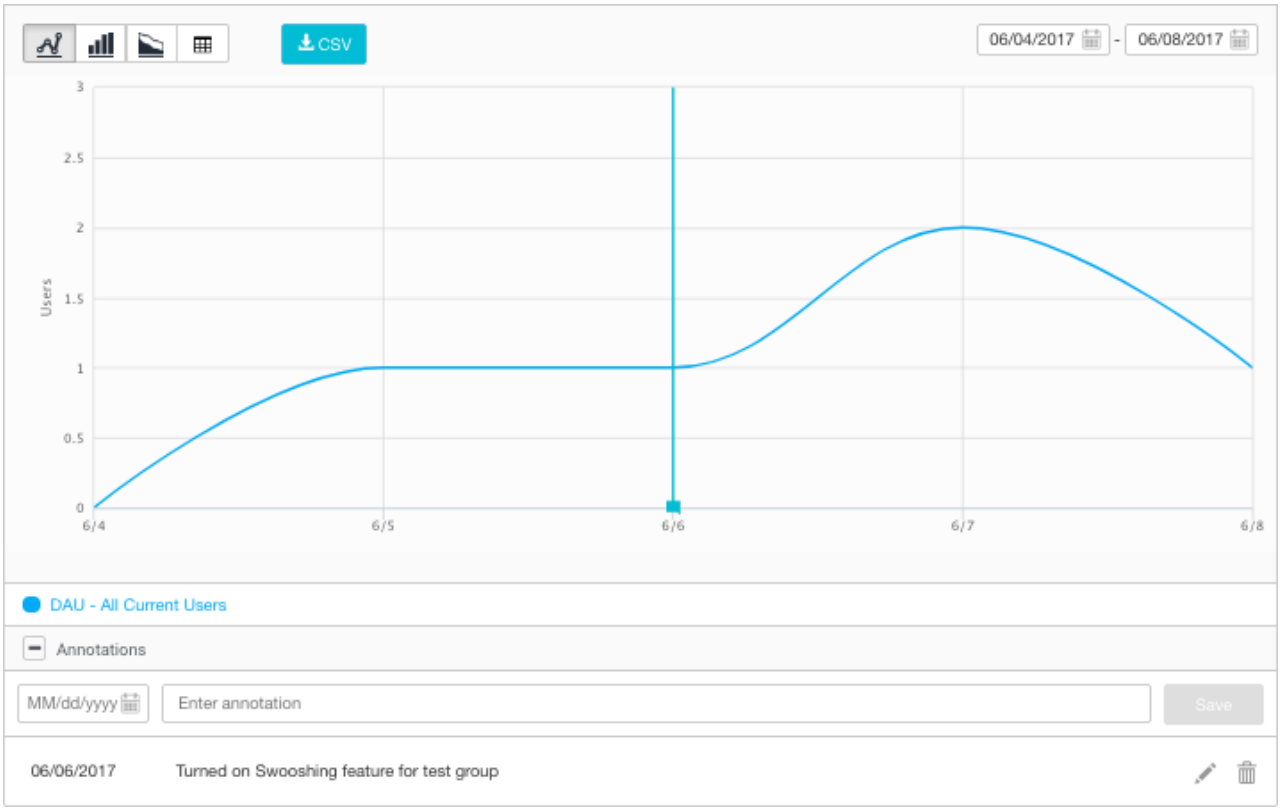


Click and drag to zoom in on a section of the chart

When you are zoomed in, the chart displays a **Reset Zoom** button, which restores the chart to display the entire date range.

Annotations

You can use annotations to mark the dates of important external factors that might affect your game's analytics data, such as game updates, marketing campaigns, and promotions. Add date-based annotations to the Data Explorer reports using the Annotations section below the report chart.



Annotation marker shown on a report chart

Annotations are project-wide, not specific to an individual report. Annotations display on all report charts whenever the Annotations section is open.

To add an annotation:

On any Data Explorer report, click the **+ Annotations** button to open the **Annotations** section (if not already open).
Click in the date box and set the date for the annotation. You can only add one annotation record for a given day.
Enter a name or descriptive text for the annotation.
Click **Save**.

2017–08–29 Page published with editorial review
New feature in Unity 2017.1

Funnel Analyzer page

Leave feedback

Funnels track player progress through linear sequences of events, tasks, or actions in your game. You can use funnels to better understand how players move through important sequences and to detect possible bottlenecks where players run into problems. Create funnel reports on the Funnel Analyzer page of your **Analytics** Dashboard.

Before creating a funnel report, you must implement the necessary Standard and Custom Events in your project. See Funnels for more information on designing, implementing, and analyzing funnels.

2017–08–29 Page published with editorial review

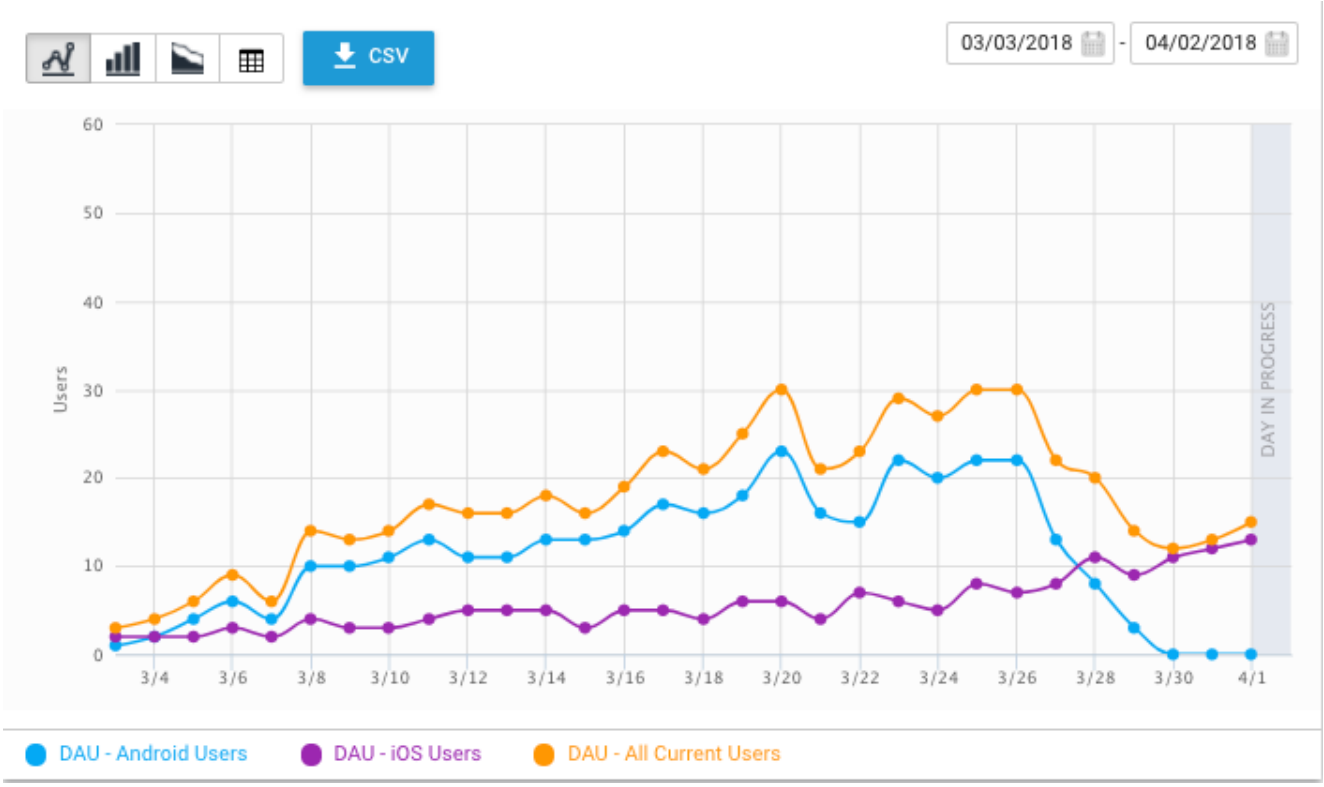
New feature in Unity 2017.1

Segment Builder

Leave feedback

Segments are subsets of your player base, split apart by key differentiators, such as country, platform, experience level, or spending patterns. View existing segments and create new ones on the **Segment Builder** page.

Use segments in the **Data Explorer** and **Funnel Analyzer** to compare metrics across segments. For example, if you publish your game on both Android and **iOS**, you can use segments to compare player behavior by platform.

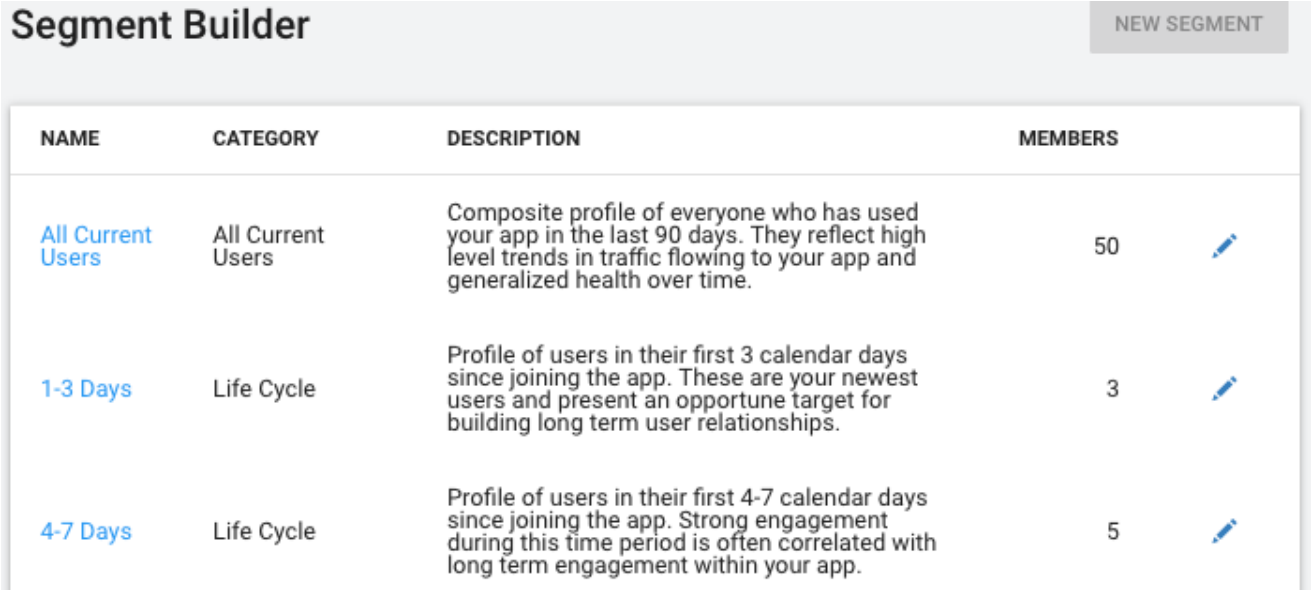


Using segments to compare DAU by platform in the Data Explorer

Unity **Analytics** defines the following categories of segments:

Life Cycle	Segments based on the number of calendar days since a player first used your app. These segments are automatically populated based on core **analytics events**.
Geography	Segments based on where the player is located in the world. These segments are automatically populated based on analysis of the player's IP address and other geolocation techniques.
Monetization	Segments based on a player's verified in-app purchases (IAP). These segments are automatically populated if you use Unity's IAP service. Otherwise, you can report verified IAP purchases using the Analytics.Transaction function. Note that IAP verification is only supported by the Apple App store and Google Play store.
Platform	Segments based on the player's platform. These segments are populated automatically based on the Unity build.
Custom	Any segments you define.

The **Segment Builder** page lists all the existing segments and shows the current segment population.



The Segment Builder page

You can change the name or delete existing segments, but you cannot change the rules that make up the segment definition. (To use different rules, create a new segment.)

Create a new segment

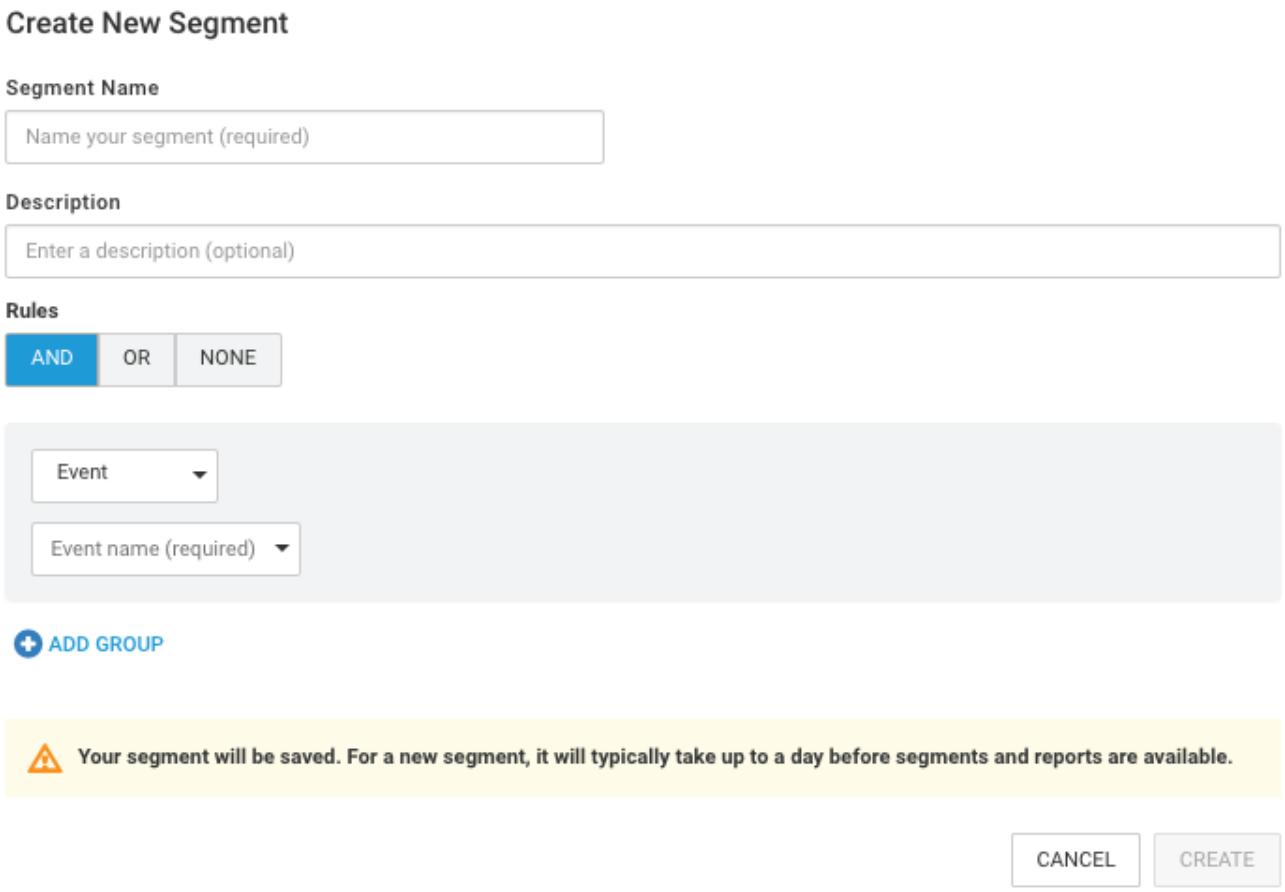
Create your own segments on the **Segment Builder** page of your Analytics dashboard.

Important: Unity Analytics evaluates the rules defined in a segment as it processes incoming data. Existing data is not re-evaluated when you create a new segment. Only data received after you create it appears in that segment in **Data Explorer** and **Funnel Analyzer** reports.

To create a segment:

On the Analytics dashboard, click the **Segment Builder** page.

Click the **+ New Segment** button.



Define one or more rules that incoming events must satisfy in order for a user to be included in the segment.
Click the **Save** button.

Segment Rules

For each custom segment, define one or more rules to determine whether a player is included in a segment. Once a player is included in a custom segment, they are considered part of that segment for up to 90 days even if no further qualifying events are received. When designing segments, take into consideration that there is no way for a player to leave a segment. For example, avoid creating a segment like “Players who have played fewer than 5 levels.” Because players do not leave the segment, they will still be included when they have played more than 5. A better approach might be a set of segments like “Players who have played 1, 2, ..., n levels” (at whatever level of detail you find useful).

You can define multiple rules for a segment. You can combine the rules such that **ALL** the rules must apply, **ANY** rule must apply, or **NONE** of the rules can apply for a player to join a segment.

The rules themselves can be derived from core, session-based Analytics events (similar to the standard segments) or can be based on Standard and Custom Events.

Core Analytics segment rules

The standard segments use rules based on the core Analytics events. You can create segment rules using these same criteria by setting the **Rule** type to any option other than **Event**. For example, if you wanted platform-based segments beyond the standard Android and iOS segments, you could choose the rule type, **User platform is_** and then choose one of the available platforms:

A rule defining a custom platform segment
Each of the rule types has its own set of options.

Standard and Custom Event segment rules

To create a segment based on **Standard** and **Custom Events**, leave the **Rule** type as **Event** and choose the event name from the **Event name** list. Only events that have already been received from your game appear in the list. Thus, before creating event-based segments, you should dispatch at least one instance of the event (which you can do while running your game in the Unity Editor). Remember that events dispatched by your game do not show up in the dashboard for several hours.

In addition to the event name, you can set one or more restrictions on the allowed parameter values. If you do not specify any parameter restrictions, then dispatching the event with any parameter values, or no parameters at all, is enough to satisfy the rule.

2017–08–29 Page published with editorial review

2018–06–04 - Removed Demographics Segments, which are no longer supported.

New feature in Unity 2017.1

Remote Settings page

Leave feedback

Remote Settings are variables that you can set on your **Analytics** Dashboard and access in your game. Manage Remote Settings on the Remote Settings page, which you can find in the **Optimization** section.

For information on implementing and using Remote Settings variables in your projects, see Remote Settings.

2017–08–21 Page published with editorial review
New feature in Unity 2017.1

Livestream page

Leave feedback

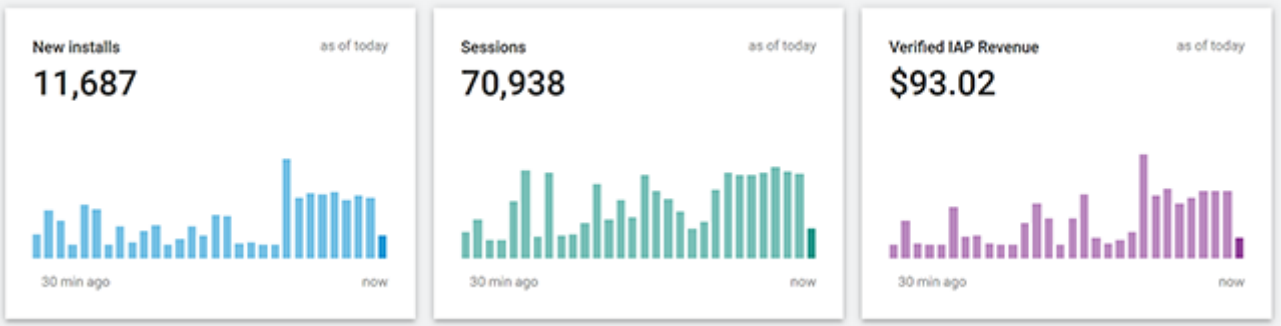
Visualize your data in real time at critical moments like game launch and during promotions.

The Livestream page samples your **Analytics** data as it happens without any other data processing. Note that other areas of the **Analytics** Dashboard do not show data until processing is complete, introducing a delay of several hours. Livestream requires a Unity Plus or Pro subscription.

The Livestream page has these sections:

Live Metrics
Activity Map
Top Country Metrics
Top Custom Events

Live Metrics

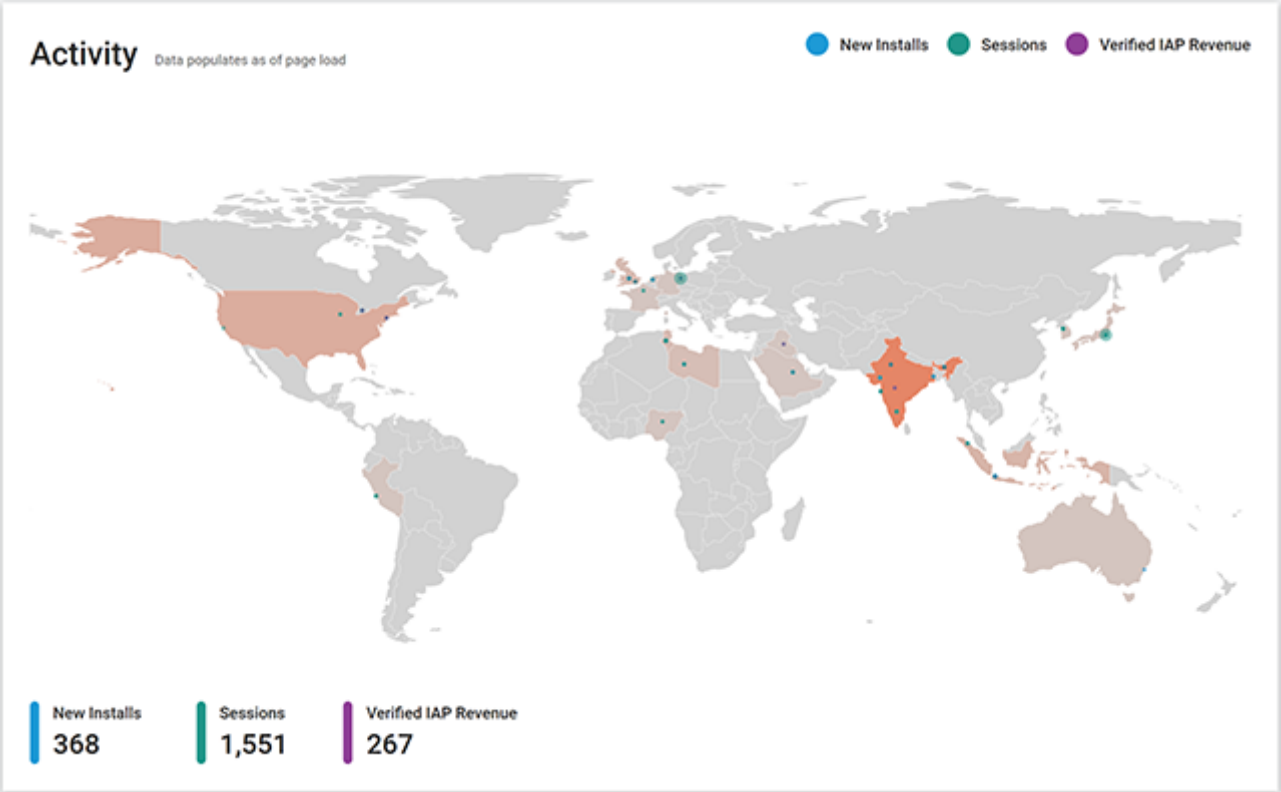


Live Metrics

The **Live Metrics** section shows charts of incoming metrics data. The numbers in the display show the current day's cumulative totals (reset at midnight GMT). The chart portions show activity over the last 5 minutes, starting when you load the Livestream page.

User Logins	The number of players who have started a new session.
New Users	The number of new players.
Verified IAP Revenue	Reported IAP revenue (after receipt verification is performed).

Activity Map



Activity Map

The **Activity Map** marks the geographic locations of incoming Analytics metrics events.

Top Country Metrics

Top 10 Countries

Data populates as of page load

Country	New Installs	Sessions	Verified IAP Revenue
India	134	462	76
United States of America	36	233	30
Indonesia	45	155	25
Tunisia	31	105	19
United Kingdom	25	104	25
Libyan Arab Jamahiriya	21	104	15
Japan	12	68	8
Germany	9	56	12
Peru	14	55	8
Netherlands	10	59	8

Top Country Metrics

The **Top Country Metrics** section shows metrics from the most active countries.

Top Custom Events

Top 10 Custom Events Data populates as of page load

Event	Count
levelStart	294
levelComplete	70

Top Custom Events

The **Top Custom Events** section shows the most common **Standard** and **Custom Events** dispatched while players use your game. The event counts are cumulative since you loaded the Livestream page.

Note that the events shown on the Livestream page are sampled for efficiency. This means that not every event is individually counted and, if you open Livestream in two different pages, the numbers and events shown can be slightly different.

2018–03–30 Page published with limited editorial review

New feature in Unity 2017.1

Added Top Country Metrics.

Raw Data Export page

Leave feedback

Export your raw event data so that you can process and visualize it using external tools.

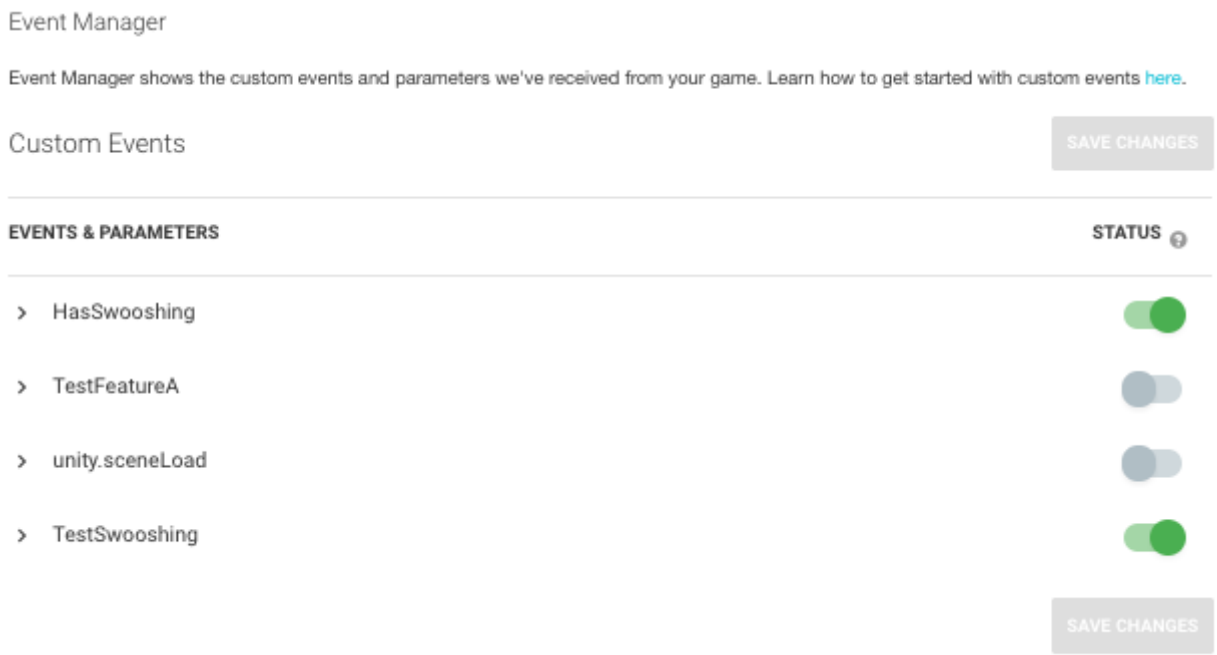
Raw Data Export requires a Unity Pro subscription. See Unity Analytics Raw Data Export for more information.

2017–08–21 Page published with editorial review
New feature in Unity 2017.1

Event Manager

Leave feedback

View a list of all the standard and custom events and parameters you have dispatched from your project. Turn off old, unused events to prevent them from appearing in new reports.

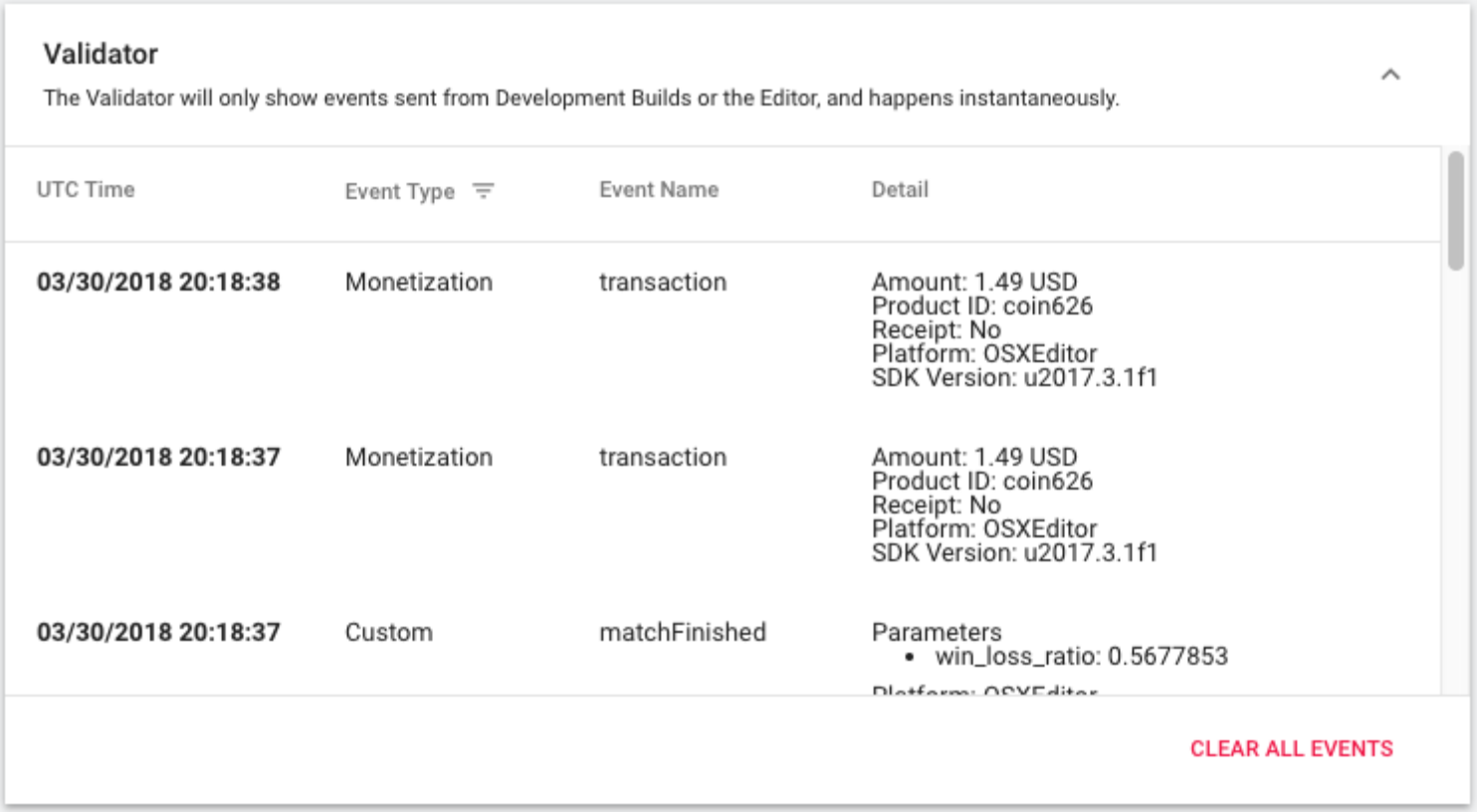


The Event Manager page showing two disabled events

When you disable an event, the Unity **Analytics** system ignores that event when processing your data. Because disabled events are ignored during data processing, they do not count toward your account event limits. If you re-enable an event, Unity **Analytics** includes newly dispatched instances of that event in subsequent reports, but any event instances dispatched while the event was disabled remain unprocessed. Note that you can download disabled events through Raw Data Export. The Event Manager only controls whether events are aggregated with the rest of your Analytics data.

Validator

The Validator displays events sent from **development builds** of your game (and from Play mode in the Unity Editor) as they are received by the **Unity Analytics** service.



Event Validator

Use the Validator window during development to verify that the **Analytics events** you send are received by the service. The events shown by the Validator on the Analytics Dashboard should match the events shown by the version of the Validator shown on the Analytics section of the Services window in the Unity Editor.

Validator in Editor

2018–03–30 Page published with editorial review

New feature in Unity 2017.1

Validator display moved to the event Manager page.

Configure page

Leave feedback

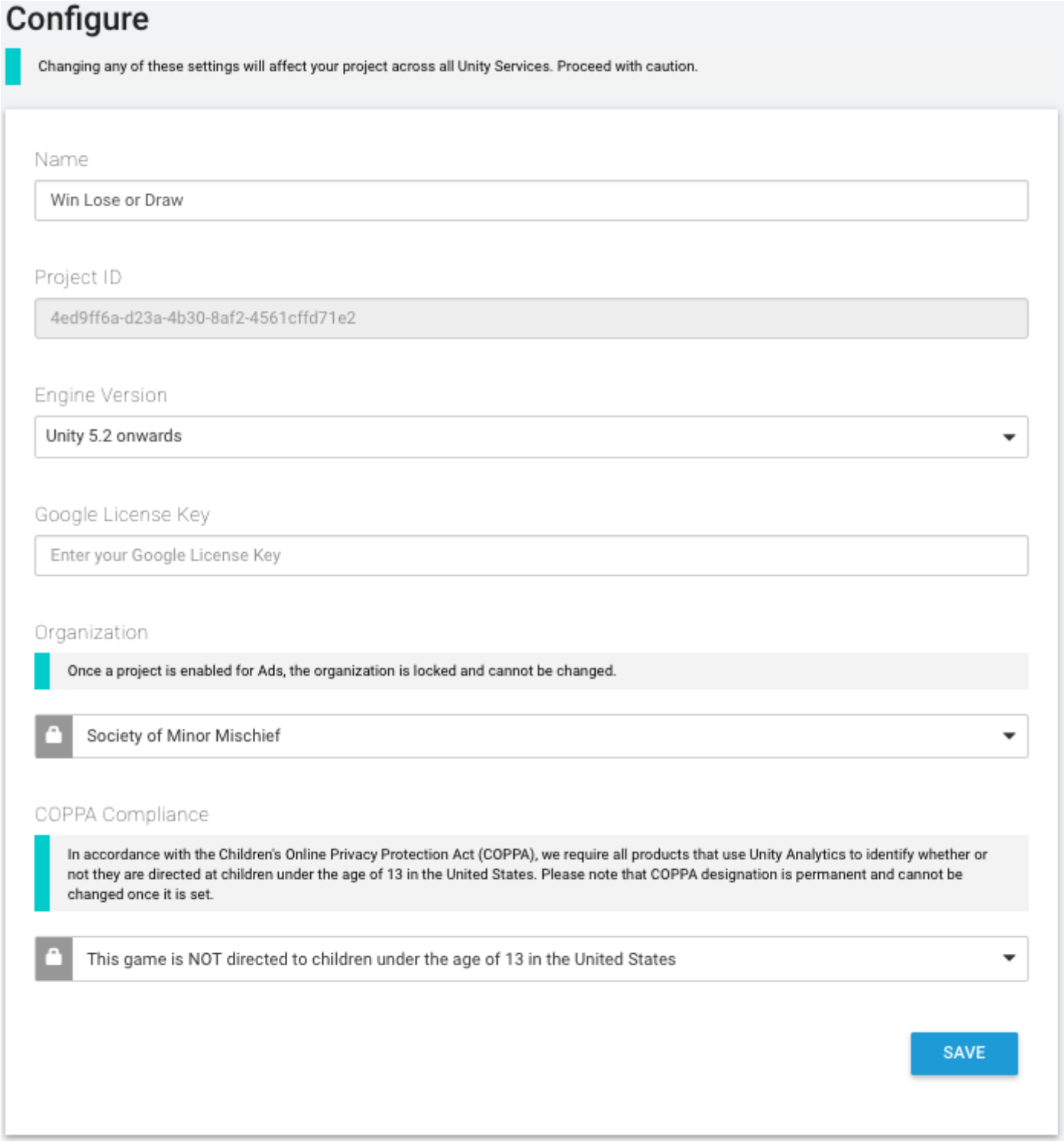
View and edit settings for individual Unity Analytics projects on the Configure page. Find the Configure page under **Settings** > **Analytics Settings** on the dashboard.

The Configure page contains these sections:

Project Service Settings
Feature Settings
Admin Settings

Project Service Settings

The Project Service Settings control aspects of a project for the **Analytics** service. As noted in the table below, some settings also affect the Ads and IAP services.



Project Service Settings section of the Analytics Configure page

Name	The name of the project. Displayed in the Unity Services Dashboards and the Unity Editor Services windows.
Project ID	A unique identifier for the project. Used to identify the project between Unity Services and the Editor.
Engine Version	Specifies the Unity Editor and Engine version in use. The value set here changes the information shown on the Integration pages.
Google License Key	Enter your Google license key if you want to validate in-app purchases (IAP) made through the Google Play store.
Organization	The name of your Unity Organization.

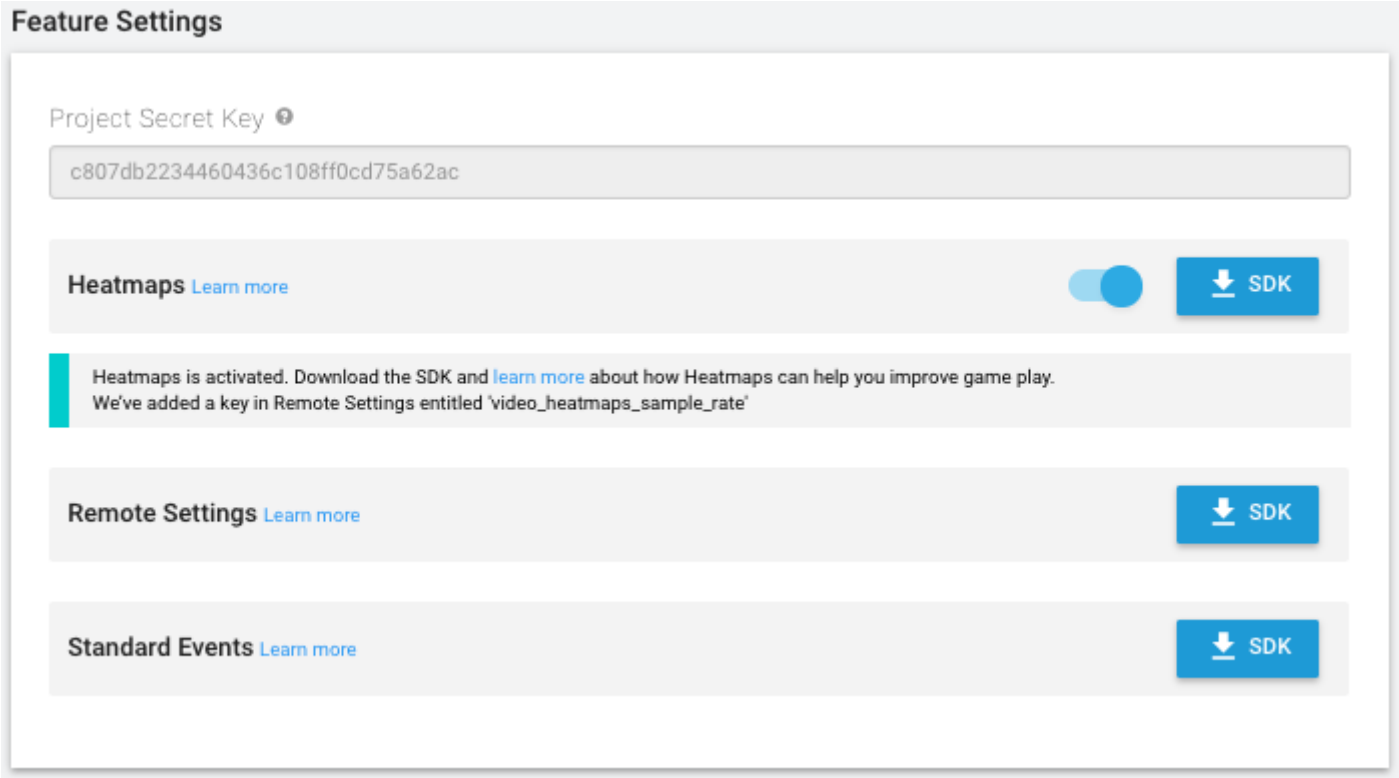
COPPA Compliance Specifies whether your project is directed at children and falls under the domain of the U.S. Children's Online Privacy Protection Act (COPPA).

When you specify that your application is subject to **COPPA**, the Unity Ads service alters the way it serves advertisements to the user to conform with the rule. Data collected by the Analytics service remains the same.

It is solely your responsibility to ensure that your application and your usage of Analytics data conforms to applicable laws (Unity cannot supply legal advice on this matter.) See COPPA Compliance for more information.

Feature Settings

The feature settings section provides information and settings for specific Analytics-related features.



Feature Settings section of the Analytics Configure page

Project Secret Key

The project secret key authorizes access to your Analytics data. Use the project secret key with the Raw Data Export REST API, as well as for Remote Settings and Heatmaps support in the Unity Editor.

Note that you should keep the **Project Secret Key** secret. Anyone possessing this value can potentially access your Analytics data.

Also note that the **Project Secret Key** was formerly known as the **Raw Data Export API Key**. The key value is the same; use the **Project Secret Key** in any place where you previously used the **Raw Data Export API Key**.

Heatmaps (Pro-only)

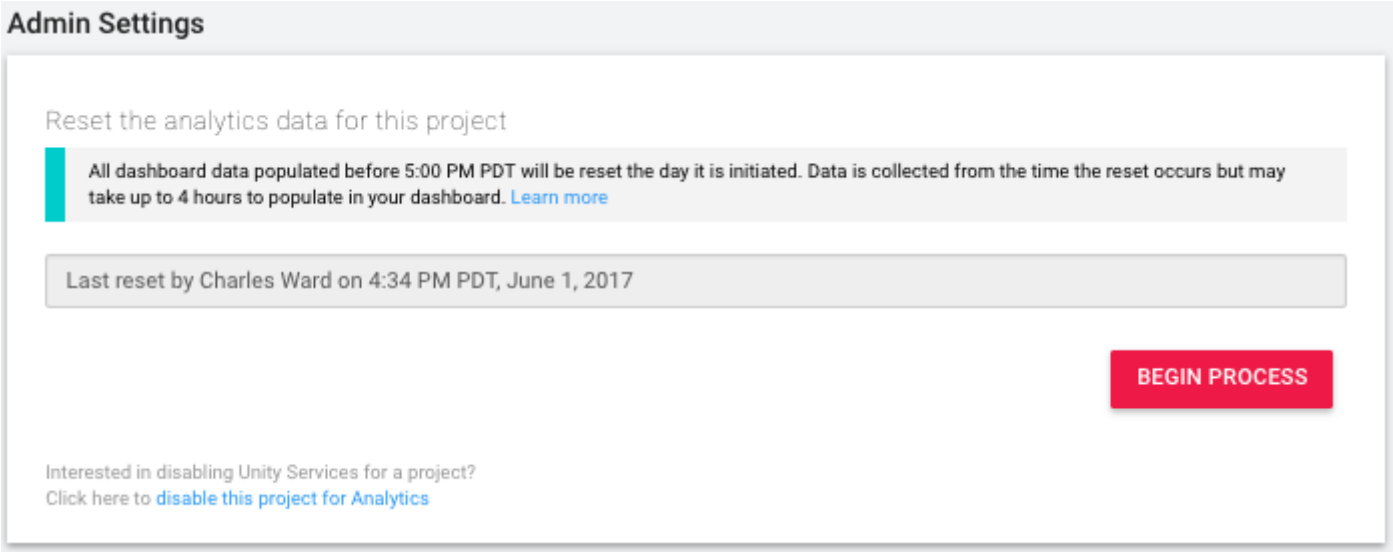
Heatmaps visualize your Analytics data in two and three dimensions by displaying graphical markers of events directly in your Editor scene.

To use Heatmaps, enable the feature on the Configure page, then click the **SDK** button to download the Heatmaps asset package.

Remote Settings	Click the **SDK** button to download the Remote Settings **asset** package.
Standard Events	Click the **SDK** button to download the Standard Events **asset package**.

Admin Settings

The Admin Settings section contains controls that let you delete existing Analytics data or disable Analytics altogether.



Admin Settings section of the Analytics Configure page

Reset Analytics Data

Resetting your Analytics data permanently deletes all existing processed data so that it no longer appears in any Dashboard reports, segments or funnels. However, your raw data is not deleted and you can still be download it using Raw Data Export.

The processed data is deleted at the time stated on the Configure page, at which point Analytics data collection, processing, and aggregation begins anew.

Click the **Begin Process** button and follow the prompts to complete the data reset.

Turn off Analytics

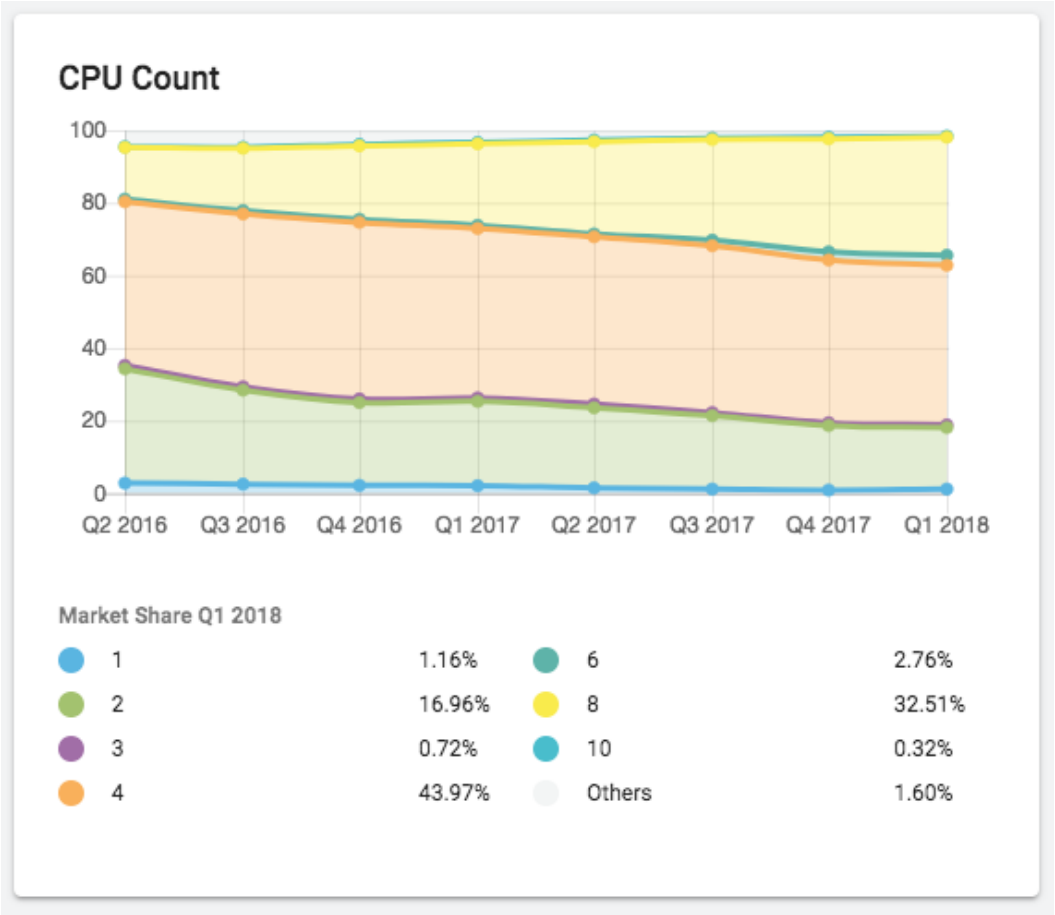
Click the **disable this project for Analytics** link to turn off **Unity Analytics** for the project. (You can re-enable Analytics for the project at any time from the Unity Editor or the Services Dashboard.)

2017–08–29 Page published with editorial review
New feature in Unity 2017.1

Market Insights

Leave feedback

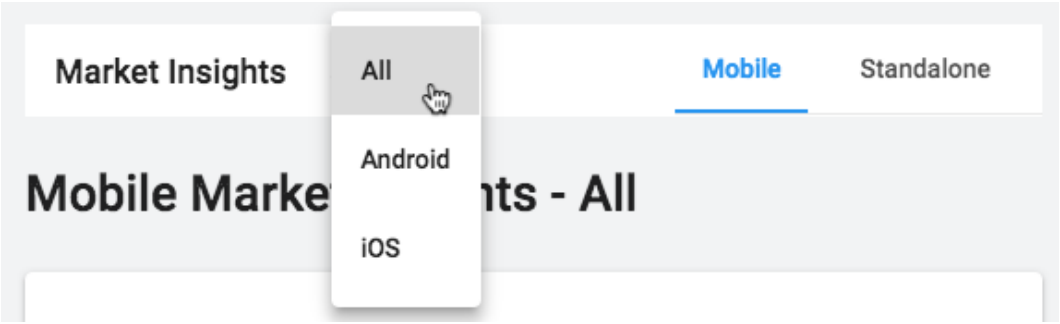
Use the Market Insights page to find trends in the computer hardware that players use to play Unity games.



The Market Insight report for Mobile CPU Count

The Market Insights page shows the relative market share of various facets of computer hardware. For example, the chart above shows the number of CPU cores a computer or device has. Each report displays the most common members of the category (after the most popular members, the market share is grouped into a single **Others** item). Unity **Analytics** compiles the report data quarterly and provides historical data for the past two years.

Data for mobile and personal computer (Standalone Player) are compiled separately. Use the **toolbar** at the top of the page to select which set of data to view.



Market Insight data set selectors

You can also look at separate data sets for Android and **iOS** on mobile, and for Windows and macOS on Standalone Players.

Report	Description
Aspect Ratio	The ratio between the long and short edges of the display screen.
CPU	The model name of the Central Processing Unit (CPU).

Report	Description
CPU Count	The number of CPU cores.
CPU Vendor	The vendor who provided the CPU. (Standalone platforms only.)
GFX Name	The model name of the Graphics Processing Unit (GPU).
GFX Vendor	The vendor who produced the GPU.
OS Version	The name and version of the operating system.
Platform	The platform name.
RAM	The amount of Random Access Memory available.
Screen	The **pixel** dimensions of the display screen.
Vendor	The vendor who produced the device. (Mobile platforms only.)

2018–08–23 Page published with no editorial review

Marked Insights page added to Analytics Dashboard in 2018.3

Analytics events

Leave feedback

Unity collects **Analytics** data by dispatching events when someone uses your application or game. Unity dispatches a set of core events automatically. You can dispatch additional events to collect richer and deeper information about how users play your game or use your application.

The types of **Analytics events** include:

Core Events: Session-based and device-based events.
Standard Events: Behavioral events with standard names and parameters for tracking user experience and player behavior.
Custom Events: Fully-customizable events whose names, purposes and parameters you define as needed.
Transaction Events: Specialized events you can use to track in-app purchase (IAP) transactions made through 3rd-party IAP APIs.
Other: Specialized events, like Heatmap events, intended for a specific purpose. Such events are usually based on Custom Events, but the APIs for sending them control the event names and parameters.

Unity dispatches Core Events automatically. You must dispatch the other types of events at the appropriate times. You can dispatch Standard and Custom Events from an Analytics Event Tracker Component or from a script.

Your Unity game or app sends Analytics events to a Unity web service. The Analytics system collects the raw events and processes them for display on your project's Analytics Dashboard. If the computer or device running the application cannot connect to the web service, Unity caches the events locally until the connection becomes available.

Note: If you have a Unity Pro subscription, you can download the raw event data directly. You can import this raw data into your own analytics tools and perform ad-hoc analyses that may not be supported by the **Unity Analytics** Dashboard.

Standard Events versus Custom Events

Standard Events are a form of Custom Event. The difference is that Standard Events are standardized. Each Standard Event has a predefined event name and a set of required parameters. Standardization provides several benefits:

Helps avoid easy-to-make coding errors, such as using slightly different names and parameters for what should be the same event.
Provides a guide for useful places in your application to dispatch an Analytics event.
Provides a known context for an event, which the Unity Analytics system can use to provide more meaningful Analytics reports.

Because of these benefits, you should use Standard Events rather than Custom Events whenever possible. Only use Custom Events when no Standard Event is appropriate.

2018–03–02 Page published with no editorial review

2018–06–04 - Service compatible with Unity 5.2 onwards at this date but version compatibility may be subject to change

New feature in 5.2

2018–06–04 - Demographics event removed

Core Events

Leave feedback

Core Events are session-based and device-based **Analytics** events. When you enable Unity **Analytics** for your project, Unity dispatches Core Events automatically. Core Events provide the basis for several metrics computed by the **Analytics** system, including:

New installs
Daily active users (DAU)
Monthly active users (MAU)
Total sessions
Sessions per user
Time spent in app
User Segments for country and platform
Revenue (when using Unity Ads and/or Unity IAP)

See Analytics Metrics, Segments, and Terminology for the definitions of these metrics.

Core Events include:

AppStart: Dispatched at the start of a new session. A new session starts when the user first launches an app or brings an app to the foreground after more than 30 minutes of inactivity.
AppRunning: Dispatched periodically while an app runs.
DeviceInfo: Dispatched the first time a user launches an app and whenever device information changes.

In addition to viewing the metrics based on core events in the Analytics Dashboard, you can download the events themselves using Raw Data Export. Note that Raw Data Export requires a Unity Pro subscription.

Note: Analytics uses other Core Events, such as **AppStop**, **AppUpdate**, and **AppInstall**, to calculate metrics, but does not expose them through Raw Data Export.

2018–06–22 Page published with editorial review

2018–03–02 - Service compatible with Unity 5.2 onwards at this date but version compatibility may be subject to change

2018–06–04 - Removed UserInfo core event.

New feature in 5.2

Standard Events

Leave feedback

Use **Standard Events** to track user experience and player behavior across five major areas:

Application: Track the usage of basic elements in your application user interface
Progression: Track player progress through your application or game
Onboarding: Track the earliest interactions players have with your application or game
Engagement: Track important actions related to social sharing, achievements
Monetization: Track revenue-related events as well as your in-game economies

The list of Standard Events covers a set of things that you might find useful to track in your games and applications. For example, most games have a concept of player progress, which could mean puzzles completed, matches played, RPG-type experience earned, or some other notion. By tracking standard progression events, you can better understand where players stop progressing and, ultimately, where they stop playing.

Most Standard Events define required and optional parameters that provide additional information about the state of your game when an **Analytics** event is sent. You can also define your own custom parameters. Setting parameters on your event allows you to filter data collected at the time the event occurred. Visualization tools for Standard and Custom Events can be viewed on the Analytics Dashboard, including **Data Explorer**, Funnel Analyzer, and Segment Builder.

Unity Analytics limits the number of Standard and Custom Events it accepts to 100 per hour, per user. In other words, if a single user playing your game manages to trigger more than 100 Standard or Custom Events within an hour, Analytics discards the additional events.

Note: To use Standard Events prior to Unity 2017.3, first download and import the Standard Events SDK from the Unity **Asset store**.

Application events

Application events track player interaction with the application outside of the game itself (for example, in the menu or cut-scene screens before or after a level). You can analyze application events to see whether players use the basic parts of your user interface as often as you expect.

Event name	Purpose
screen_visit	Player opened a screen in the **UI**, such as a high score or settings screen.
cutscene_start	Player began watching a cinematic cutscene.
cutscene_skip	Player skipped past a cinematic cutscene.

Progression events

Progression events track players as they advance through your game. Analyze standard progression events to monitor player progress in your game.

Event name	Purpose
game_start	Player began a new game (useful for games with a distinct beginning and ending).
game_over	Player ended the game (useful for games with a distinct beginning and ending).
level_start	Player started a level.
level_complete	Player successfully completed a level.
level_fail	Player lost a level.

Event name	Purpose
level_quit	Player quit out of a level.
level_skip	Player skipped past a level.
level_up	Player increased in rank or RPG-style experience level.

Onboarding events

Onboarding events track the first-time user experience (FTUE). Analyze standard onboarding events to monitor the effectiveness of your tutorials.

Event name	Purpose
first_interaction	Player completed any interaction after opening the game for the first time.
tutorial_start	Player began a tutorial.
tutorial_step	Player passed a milestone in a tutorial.
tutorial_complete	Player completed a tutorial.
tutorial_skip	Player skipped a tutorial.

Engagement events

Engagement events help you understand whether your players are engaging with your game, and whether they're performing actions highly correlated with retention and virality.

Event name	Purpose
push_notification_enable	Player enabled push notifications.
push_notification_click	Player responded to a pushed message.
chat_msg_sent	Player sent a chat message.
achievement_unlocked	Player completed an achievement.
achievement_step	Player completed a milestone towards an achievement.
user_signup	Player connected with a social network.
social_share	Player shared something such as an invite or gift through a social network.
social_accept	Player accepted something shared through a social network.

Monetization events

Monetization events track income and game economy to help you balance your resources and improve sources of revenue.

Event name	Purpose
store_opened	Player opened a store.
store_item_click	Player selected an item in a store.
iap_transaction	Player spent real-world money to make an in-app purchase.
item_acquired	Player acquired a resource within the game.
item_spent	Player expended an item within the game.
ad_offer	Player had an opportunity to watch an ad.
ad_start	Player started watching an ad.
ad_complete	Player finished watching an ad.
ad_skip	Player skipped an ad before completion.
post_ad_action	Player completed an action prompted by an ad.

Scripting Standard Events

To dispatch a Standard event from a script, add the following namespace:

```
using UnityEngine.Analytics;
```

Note: When using the Standard Events SDK from the Unity **Asset** store, use the namespace `UnityEngine.Analytics.Experimental` instead.

Now you can dispatch Standard Events using the AnalyticsEvent class. Each Standard Event has a corresponding static function. For example, to send the game_start event, call the following function:

```
AnalyticsEvent.GameStart();
```

Many Standard Events have required parameters, and several have optional parameters. You can also add your own additional parameters in a dictionary. For example, if you wanted to track what screens players visit using the ScreenVisit event, but also wanted to track which screen they visited from, you can pass that information in a dictionary object:

```
public static void ReportScreenVisit(string screenName, string currentScreenName)
{
    AnalyticsEvent.ScreenVisit(screenName, new Dictionary<string, object>
    {
        { "from_screen", currentScreenName }
    });
}
```

The total number of required parameters, optional parameters (only counting those that you use) and additional parameters cannot exceed ten. In other words, if an event function has 2 required parameters and you set 1 out of 2 optional parameters, you can pass a dictionary containing up to seven additional key-value pairs to the function. If you set both of the optional parameters in the same event, then your dictionary could only contain six key-value pairs.

Standard Event code example

The following class provides an example that uses the standard MonoBehaviour event functions to dispatch the Standard Events related to starting and ending levels.

The Awake function dispatches the level_start event. The example provides both the level name and the level index, but only one is required. The example also uses the name and index from the Scene object, but you can use different values.

The OnDestroy function dispatches a level_complete, level_fail, level_skip, or level_quit event, based on the current level state value (which, in a real game, would be set by other code in the level using the `SetLevelPlayState()` function defined in this example). The example adds some additional parameters to these events using a dictionary called "customParams".

```
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;
using UnityEngine.Analytics;

public class LevelEventManager : MonoBehaviour {
    public enum LevelPlayState {InProgress, Won, Lost, Skip, Quit}

    private Scene thisScene;
    private LevelPlayState state = LevelPlayState.InProgress;
    private int score = 0;
    private float secondsElapsed = 0;
    private int deaths = 0;

    void Awake () {
        thisScene = SceneManager.GetActiveScene();
        AnalyticsEvent.LevelStart(thisScene.name,
                                  thisScene.buildIndex);
    }

    public void SetLevelPlayState(LevelPlayState newState){
        this.state = newState;
    }

    public void IncreaseScore(int points){
        score += points;
    }

    public void IncrementDeaths(){
        deaths++;
    }

    void Update(){
        secondsElapsed += Time.deltaTime;
    }

    void OnDestroy(){
        Dictionary<string, object> customParams = new Dictionary<string, object>()
```

```
        customParams.Add("seconds_played", secondsElapsed);
        customParams.Add("points", score);
        customParams.Add("deaths", deaths);

        switch(this.state){
        case LevelPlayState.Won:
            AnalyticsEvent.LevelComplete(thisScene.name,
                                         thisScene.buildIndex,
                                         customParams);
            break;
        case LevelPlayState.Lost:
            AnalyticsEvent.LevelFail(thisScene.name,
                                     thisScene.buildIndex,
                                     customParams);
            break;
        case LevelPlayState.Skip:
            AnalyticsEvent.LevelSkip(thisScene.name,
                                     thisScene.buildIndex,
                                     customParams);
            break;
        case LevelPlayState.InProgress:
        case LevelPlayState.Quit:
        default:
            AnalyticsEvent.LevelQuit(thisScene.name,
                                     thisScene.buildIndex,
                                     customParams);
            break;
        }
    }
}
```

Note that this example only uses the custom parameter dictionary created in `OnDestroy()` once. However, when you dispatch an event that uses such a dictionary frequently, you should create a single instance of the dictionary and reuse it rather than creating a new dictionary each time. Reusing the dictionary object reduces memory allocations that must be garbage collected in a future frame. See Analytics Event Parameters for more information.

2018–03–02 Page published with no editorial review

2018–03–02 - Service compatible with Unity 5.2 onwards at this date but version compatibility may be subject to change

New feature in 5.2

Custom Events

Leave feedback

Custom Events can be any specific in-game action your user performs. They allow you to track the player behavior that Unity **Analytics** does not track automatically, and for which there is no **Standard Event**. Each Custom Event can have up to ten parameters. There are no required parameters.

In general, you should only using Custom Events when no suitable Standard Event is defined. Because Standard Events have a defined context, Unity can provide better analysis and visualization support than for Custom Events.

You can send a Custom Event to the Analytics Service using the Analytics Event Tracker **component**. Just select **Custom** instead of a Standard Event. You can also send Custom Events using code.

2018–03–02 Page published with no editorial review 2018–03–02 - Service compatible with Unity 5.2 onwards at this date but version compatibility may be subject to change

New feature in 5.2

Standard Events added as Unity package

Custom Event scripting

Leave feedback

As an alternative to using the AnalyticsEventTracker **component**, you can send Custom Events directly via script by calling AnalyticsEvent.Custom:

```
// Reference the Unity Analytics namespace
using UnityEngine.Analytics;

//  Use this call for wherever a player triggers a custom event
AnalyticsEvent.Custom(string customEventName, IDictionary<string, object> eventData);
```

Analytics.CustomEvent Input Parameters

Name	*Type*	*Description*
customEventName	string	Name of the custom event. The name cannot include the prefix “unity”, which is reserved for internal Unity **Analytics** events.
eventData	dictionary	Additional parameters sent to **Unity Analytics** at the time the custom event was triggered. eventData keys cannot include the prefix “unity”, which is reserved for internal Unity **Analytics events**.

The following example sends a Custom Event when a player finds a secret location in a level:

```
public void ReportSecretFound(int secretID){
    AnalyticsEvent.Custom("secret_found", new Dictionary<string, object>
    {
        { "secret_id", secretID },
        { "time_elapsed", Time.timeSinceLevelLoad }
    });
}
```

2018–03–02 Page published with no editorial review

2018–03–02 - Service compatible with Unity 5.2 onwards at this date but version compatibility may be subject to change

New feature in 5.2

Standard Events added as Unity package.

Analytics Event Tracker

Leave feedback

Use the **Analytics Event Tracker component** to send both Standard and Custom Events. While you can send these events using a script, in some cases it can be more convenient to set up the event triggers and parameters directly from the **Inspector** window of the Unity Editor.

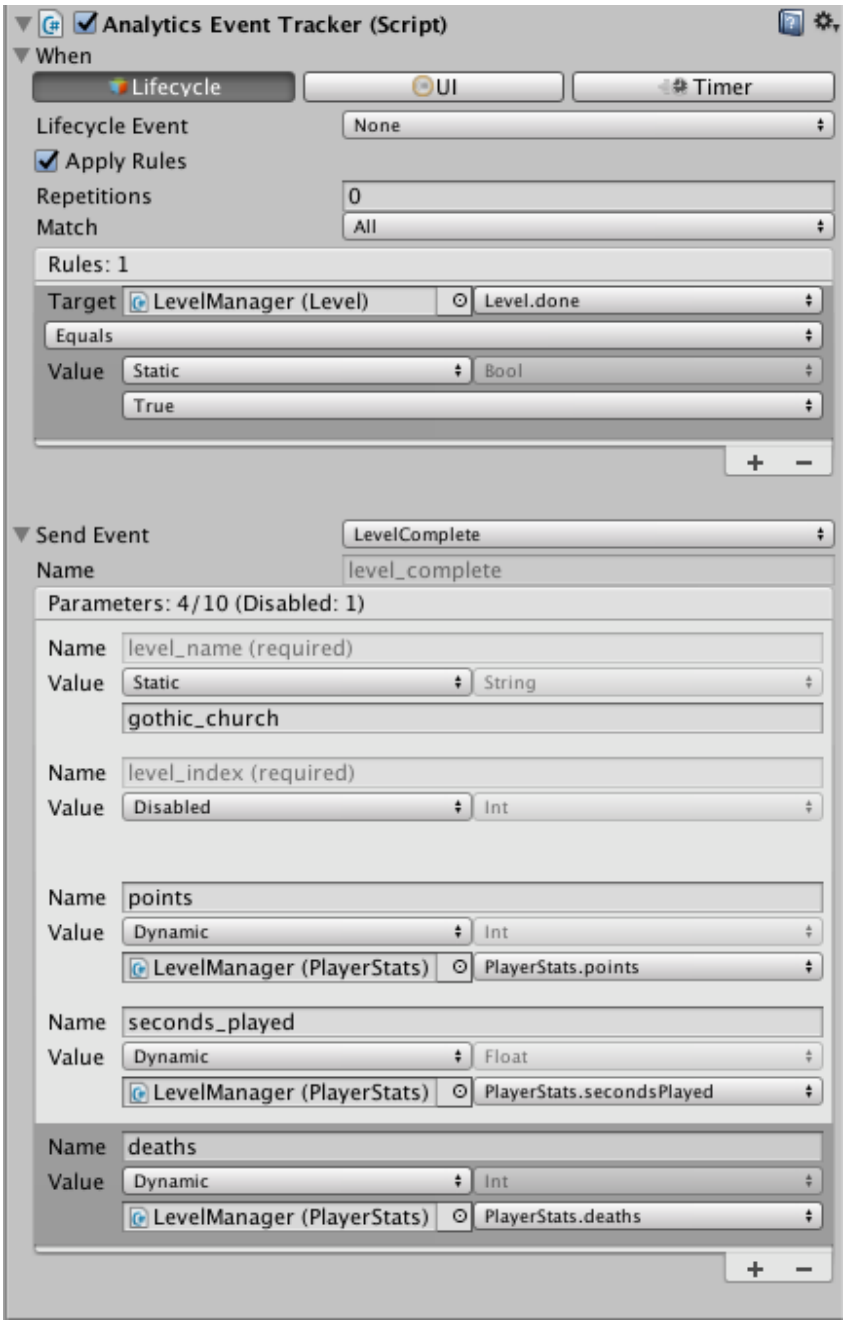
Attach an **Analytics** Event Tracker component to any **GameObject** to send a Standard or Custom event. Use a separate tracker component for each different type of **Analytics** event that you send.

The Analytics Event Tracker component has two main sections:

When: Specifies when to dispatch the event.
Send Event: Specifies which event to send, and defines the required and optional parameters.

When to send an event

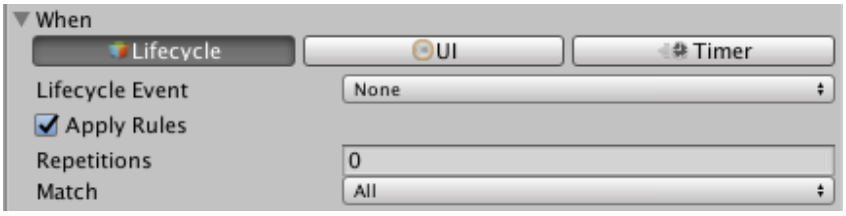
There are three ways to trigger the tracker component to send an event: **Lifecycle Event**, **UI**, and **Timer**.



Analytics Event Tracker component

Choose **Lifecycle Event** to dispatch the Analytics event when a specific `MonoBehaviour` lifecycle event (such as `Start` or `OnEnable`) occurs.

Choose **UI** if you are using a UI event to trigger the Analytics event. You must set up the UI event separately. For example, to dispatch an event when a button is clicked, you could add the `AnalyticsEventTracker.TriggerEvent` to the button's `OnClick()` list:



Specifying when to report an event to the Analytics service

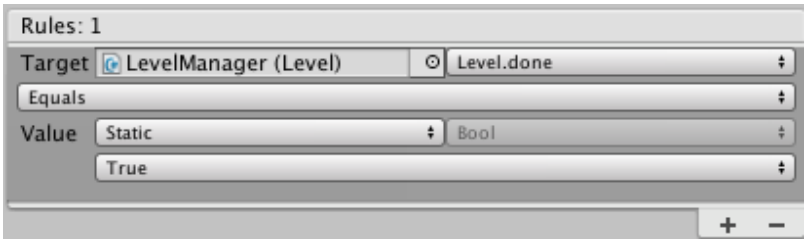
Choose **Timer** to dispatch the Analytics event after a specified delay or at regular intervals (measured in seconds). When using a timer to dispatch the event, remember that there is a limit of 100 events per hour per player, so you could quickly reach your application event limit.

Property:	*Function:*
Trigger Type	Specifies the type of game event that triggers the Analytics event to be dispatched.
Lifecycle Event	Dispatch when a specific MonoBehaviour event occurs:
Awake	Dispatch when Unity invokes this GameObject's `Awake()` method.
Start	Dispatch when Unity invokes this GameObject's `Start()` method.
On Enable	Dispatch when Unity or your game code enables this GameObject.
On Disable	Dispatch when Unity or your game code disables this GameObject.
On Application Pause	Dispatch when Unity pauses the application.
On Application Unpause	Dispatch when Unity unpauses the application.
On Destroy	Dispatch when Unity destroys the object.
None	The Analytics event is not dispatched automatically. You must call the `AnalyticsEventTracker.TriggerEvent()` method.
UI	Dispatch when a UI event occurs, such as a button press. Hook up the `AnalyticsEventTracker.TriggerEvent()` method to a suitable UI event, such as `Button.OnClick()`.
Timer	Dispatch after a time interval:
Initial Time	The interval after which the Analytics event is dispatched. The interval starts when Unity invokes the `Start()` method of the tracker object.
Poll Time	The interval at which the Analytics event is repeatedly dispatched. Use Rules to limit the number of repetitions. Set to zero for no repetitions.

Rules

You can define rules that constrain whether the tracker dispatches the Analytics event when it would otherwise be triggered. For example, if you only want to send an event when the player pushes a button and some other condition is true, you could hook up a UI trigger to the button click event and define one or more rules to specify the condition.

A rule compares the value of the field of one object to either a fixed value or the value of another field. You can use any of the typical comparison operators, such as Is Equals, Is Greater Than, and so on, as well as a couple that aren't so typical, namely Is Between and Is Between Or Equal To.



Defining rules for event reporting

If you create more than one rule, you can choose a **Match** type to specify how the rules are combined.

In addition to the comparison rules, you can choose the maximum number of times an Analytics event can be sent by setting the **Repetitions** property. Set to zero for no limit. The repetition limit applies for the lifetime of this

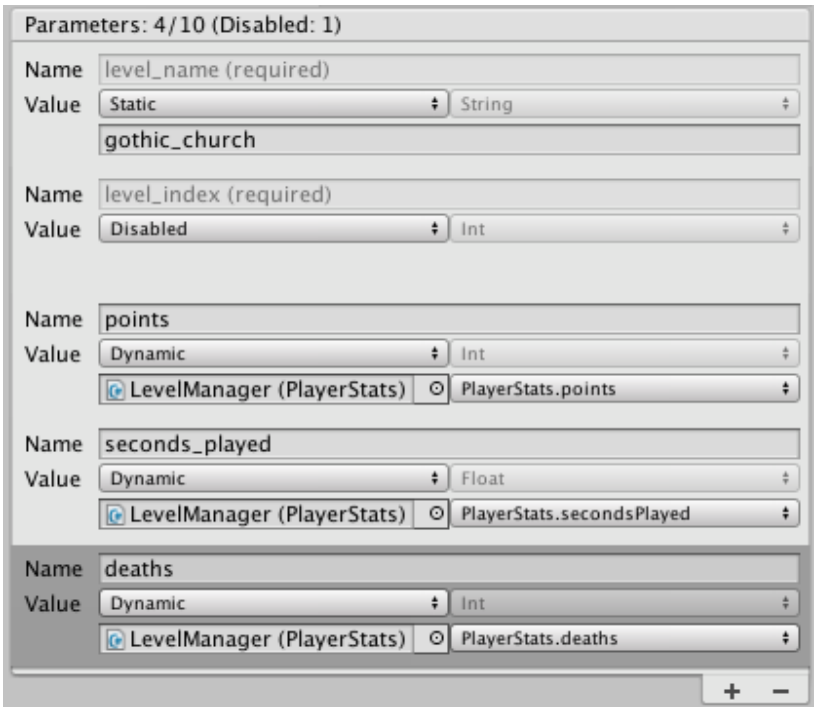
instance of the Analytics Event Tracker component. For example, if you put the component on a **Prefab** object and use more than one instance of that **prefab** in your game, each **prefab** has an independent repetition count.

Check the **Apply Rules** box to reveal the rule section of the Analytics Event Tracker component.

Property:	*Function:*
Apply Rules	Only dispatch the Analytics event if the specified rules are satisfied.
Repetitions	The maximum number of times the Analytics event will be dispatched during the GameObject's lifetime. (If the player repeats a level, for example, the events can be dispatched again.) Leave at zero for unlimited repetitions.
Match	Whether **All** rules must match, **Any** rule may match, or **None** of the rules can match in order for the tracker to dispatch the Analytics.
Rule List	Click **+** or **-** to create and delete rules. Each rule has the form: *target operator value* where *target* is a field of some MonoBehaviour object, *value* is either a static value or another field of a MonoBehaviour object, and *operator* is the type of comparison to perform.
Target Object	The object containing the field you want to test with the rule.
Target Field	The field of the object to test. You can compare fields and properties with primitive data types (float, int, bool, and string).
Comparison	How to compare the target field to the test value. For some comparisons, like *Is Between*, you specify two comparison values instead of one.
Value	Use **Static** to compare the target field to a fixed value. Use **Dynamic** to compare the target field to another field.
Value Data Type	One of the supported, primitive data types: float, int, bool, or string. When you set either a target or a value field, the tracker sets the data type to match that field.
Value Object	When using **Dynamic**, this is the object containing the field or property to compare to the target field. This property is only visible when the **Value** is **Dynamic**.
Value Field	The field to compare to the target field. The tracker compares the current values of the two fields at the time the event is to be dispatched. This property is only visible when the **Value** is **Dynamic**.
Static Value	When using **Static**, the fixed value to compare to the target field. This property is only visible when the **Value** is **Static**.

Parameters

You can define up to ten parameters to send to the Analytics Service as part of an Analytics event. Many **Standard Events** have standard required and optional parameters, which automatically appear in the list when you select the event in the Analytics Event Tracker component. The required and optional parameters do count against the ten parameter limit, but you can set optional parameters to **Disabled** to remove them. See Analytics Event Parameters for additional information.



Setting event parameters

You can assign a fixed value to a parameter, which the tracker uses every time it dispatches the event. Alternately, you can assign a dynamic value by hooking the parameter to a field or property of a MonoBehaviour object. When you use a dynamic value, the tracker assigns the value of the associated field to the parameter when it dispatches the event.

Property:	*Function:*
Name	The name of the parameter. The name of the standard parameters cannot be changed.
Parameter List	Click **+** or **-** to create and delete parameters. You cannot delete the standard parameters of an Analytics event, but you can disable the optional ones. Disabled parameters are not counted against the ten parameter limit (and are not sent to the Analytics Service).
Value	Use **Static** to assign a fixed value for this parameter. Use **Dynamic** to assign the current value of an object's field or property. Use **Disabled** to suppress optional standard parameters.
Value Data Type	One of the supported, primitive data types: float, int, bool, or string. When using **Dynamic**, the tracker sets the data type to match the specified field.
Value Object	When using **Dynamic**, the object containing the field or property to assign to the parameter when the Analytics event is dispatched. This property is only visible when the **Value** is **Dynamic**.
Value Field	The field or property containing the value to assign to the parameter. This property is only visible when the **Value** is **Dynamic**.
Static Value	When using **Static**, the fixed value to assign to the parameter. This property is only visible when the **Value** is **Static**.

2018–03–02 Page published with no editorial review

2018–03–02 - Service compatible with Unity 5.2 onwards at this date but version compatibility may be subject to change

New feature in 5.2

Analytics event parameters

Leave feedback

Both Standard and Custom events allow you to send additional information to the **Analytics** Service as part of an event. The only difference between Standard and Custom Events is that most **Standard Events** have required or optional parameters which take precedence over any custom parameters.

Pass your custom parameters to the appropriate AnalyticsEvent function in the form of a `Dictionary<string, object>` instance. The keys of this dictionary are the parameter names, and the values are the parameter values. When creating this dictionary, maintain consistent key names and data types for each parameter in your event data, both within a single version of your game as well as from version to version. For example, don't send a level name parameter as a number sometimes and as a string at other times. Doing this can make your data difficult to interpret.

Note: Do not begin the key names of the custom parameter dictionary with the prefix "unity", which is reserved for internal Unity **Analytics events**.

Unity serializes values sent to the Analytics service. It parses number characters as numbers even if the data type you add to the dictionary is a string. In other words, adding the string "*51*" to the parameter dictionary is equivalent to adding the number *51*.

You can pass up to ten parameters with an event. For Standard Events, this limit includes the required parameters and any optional parameters to which you assign a value (unused optional parameters don't count against this limit). In addition, the length of an individual key name and string value passed to the event must not exceed 100 characters and the total length of all key names and string values must not exceed 500 characters.

For efficiency, you can create the parameter dictionary instance as a class member and reuse the dictionary each time you dispatch the event. Reusing the dictionary object avoids allocating memory each time you send the event. This reduces the memory allocations that the C# garbage collector needs to clean up. The more frequently you dispatch an event in a **Scene**, the more benefit this technique provides. The following example defines a class that dispatches a Custom event. The class defines a parameter dictionary as an instance variable and sets the parameter values each time it sends an event.

```
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Analytics;

public class MyCustomAnalyticsEvent : MonoBehaviour {
    private const string Name = "my_custom_event";

    private Dictionary<string, object> parameters
        = new Dictionary<string, object>();

    void Start(){
        // Define parameters with default values
        parameters.Add("character_class", "Unknown");
        parameters.Add("health", 0);
        parameters.Add("xp", 0);
```

```
        parameters.Add("world_x", 0);
        parameters.Add("world_y", 0);
        parameters.Add("world_z", 0);
    }

    public bool Dispatch(string characterClass,
                         int health,
                         int experience,
                         Vector3 location){

        // Set parameter values for a specific event
        parameters["character_class"] = characterClass;
        parameters["health"] = health;
        parameters["xp"] = experience;
        parameters["world_x"] = location.x;
        parameters["world_y"] = location.y;
        parameters["world_z"] = location.z;

        // Send event
        AnalyticsResult result
            = AnalyticsEvent.Custom(Name, parameters);
        if(result == AnalyticsResult.Ok){
            return true;
        } else {
            return false;
        }
    }
}
```

You can use the same technique for the custom parameters you send with Standard events.

2018–03–02 Page published with no editorial review

2018–03–02 - Service compatible with Unity 5.2 onwards at this date but version compatibility may be subject to change

New feature in 5.2

Funnels

Leave feedback

Funnels help you track whether players make it through linear sequences in your game. A funnel reveals what percentage of your players make it through each step of the sequence as well as the sequence as a whole. Common sequences to track with a funnel include:

First-time user experience — track how new players make it through your on-boarding sequence
Game Progression — track how players progress through the major milestones in your game, such as level completion
Purchase **UI** — uncover impediments to making a purchase

Funnels are based on Standard and Custom events. Unity **Analytics** evaluates each step in a funnel according to whether the defined standard or custom event has been dispatched with the correct parameter values. In order for users to enter and progress through a funnel, they must complete the funnel steps in the precise order those steps appear in the funnel.

Create funnels on your Analytics Dashboard.

Important*: **Unity Analytics evaluates the rules defined in a funnel as it processes incoming data. Existing data is not re-evaluated when you create a new funnel. Only data received after you create a funnel appears in the Funnel Analyzer report.

Create a new funnel

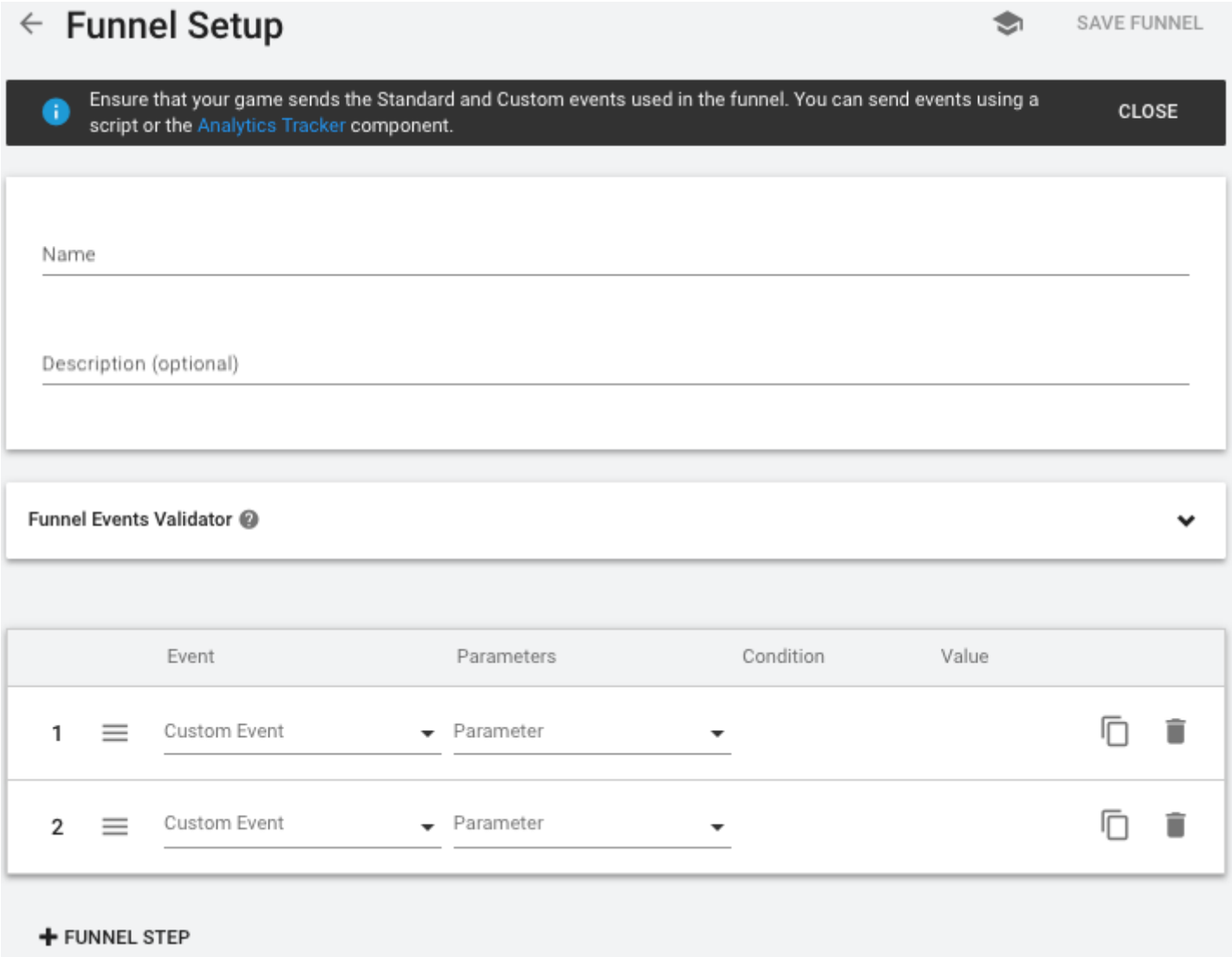
Before creating a funnel, you should first:

Add the necessary **Standard** or **Custom Events** to your game.
Run your game so that at least one instance of each event is dispatched to the Unity Analytics system. You can perform this step while running your game in the Unity editor.
Wait for Unity Analytics to process the events (about 8–16 hours).

After dispatching at least one instance of the Standard and Custom events that you want to include in the funnel, and waiting until those events are processed, follow these steps to create the funnel rules on the Analytics Dashboard:

On the Unity Analytics dashboard, click **Funnel Analyzer**.

Click **Create Funnel**.



Enter a **Name** and, optionally, a **Description** for your funnel.

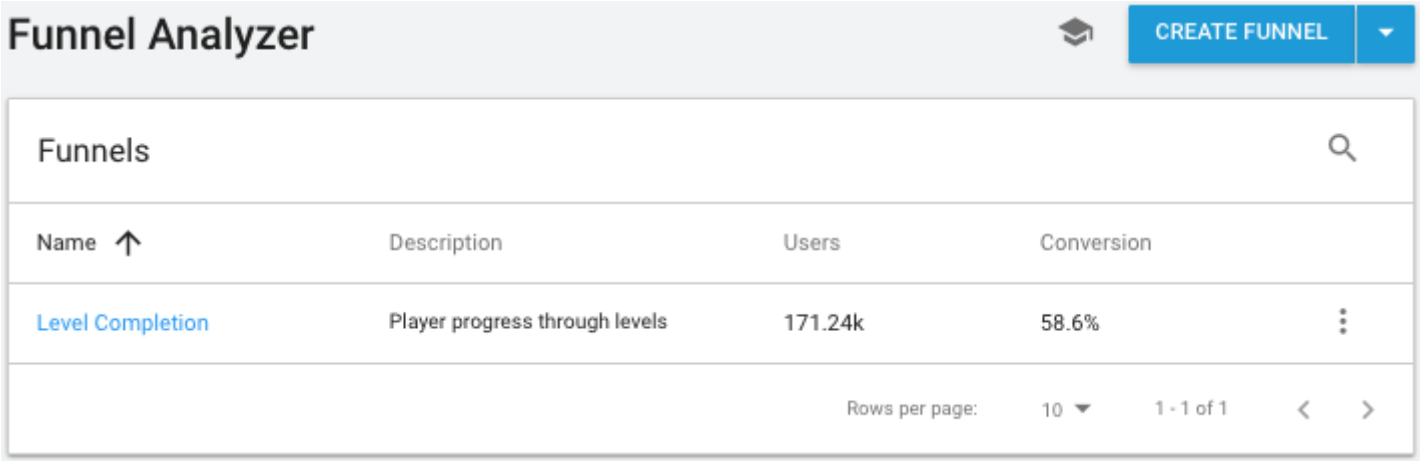
For each step in the linear sequence of events you want to track and analyze:
Click **+ Funnel Step**. (The first two steps are created automatically, since a funnel always requires comparing at least two phases of progression.)
Select the **Standard** or **Custom Event** name. Only events that have been dispatched by an instance of your game and processed are listed.
Set rules for the **Parameters** (optional).
The parameters dispatched with an event must satisfy the rules you define in order for that player to successfully complete that step in the funnel. If you do not define parameter rules, simply dispatching the event completes the step.
Click **Save**.

While you are creating the funnel steps, you can use the **Delete**, **Duplicate** and **Rearrange** icons at the top of each step to make changes.

Note: After you've clicked **Save**, funnel steps cannot be modified. However, you can create a copy of the funnel and modify the steps of the copy. The reason for this restriction is that changing the rules of an existing funnel would invalidate any existing data included in that funnel.

Reading a Funnel Analyzer report

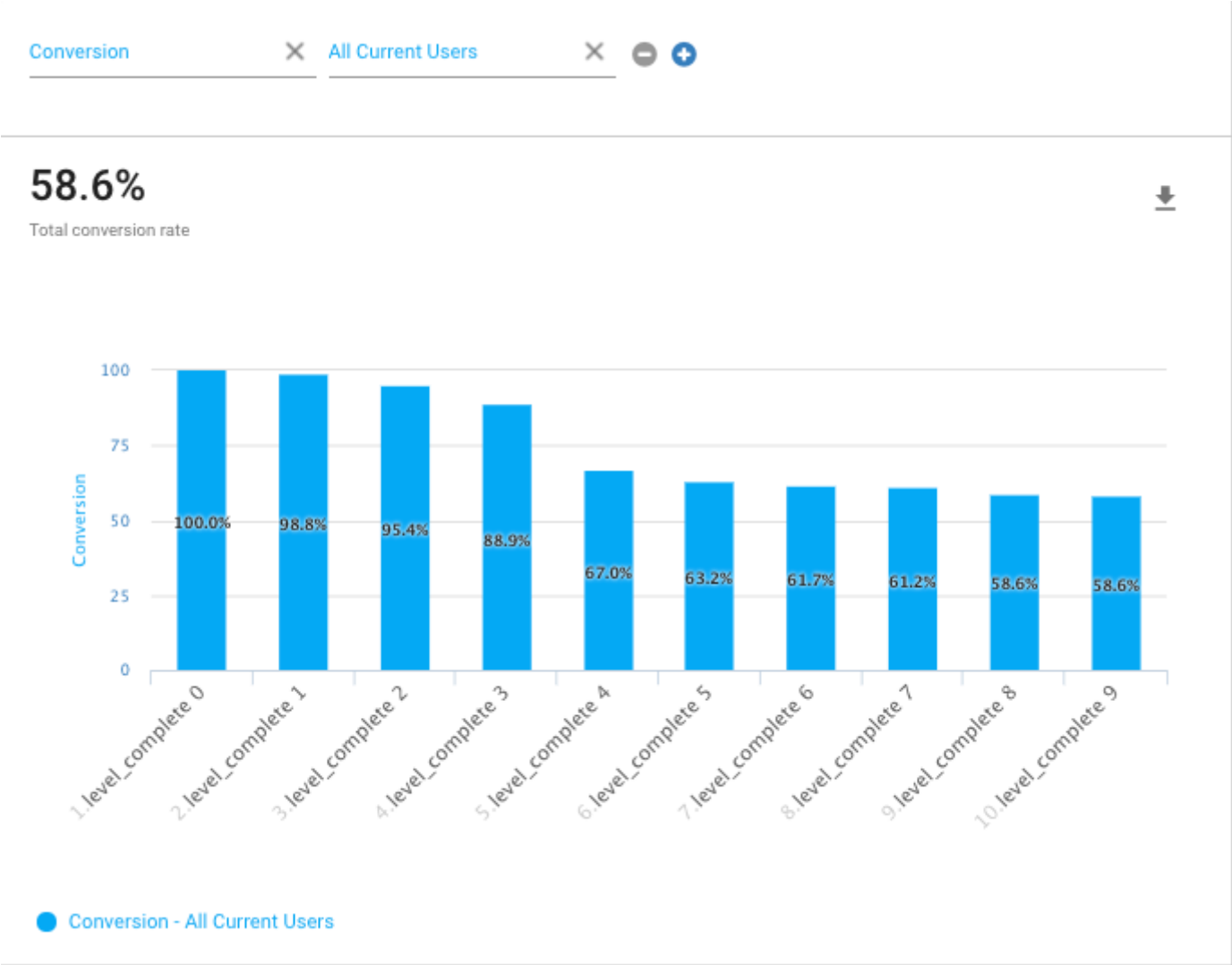
The **Funnel Analyzer** page of your Analytics Dashboard lists the funnels that you have created.



Check the **Members** column to see how many players have entered the funnel. A **Funnel Analyzer** report has two sections: the **Funnel** chart and the **Drilldown** section, which provides more detailed information about the funnel data.

Funnel conversion chart

The **Funnel** conversion chart provides an immediate overview of how players have progressed through the sequence of steps tracked by the funnel. Unexpected drop-offs between steps indicate good places to investigate game play, user interface, or other issues that block players from continuing to the next step.

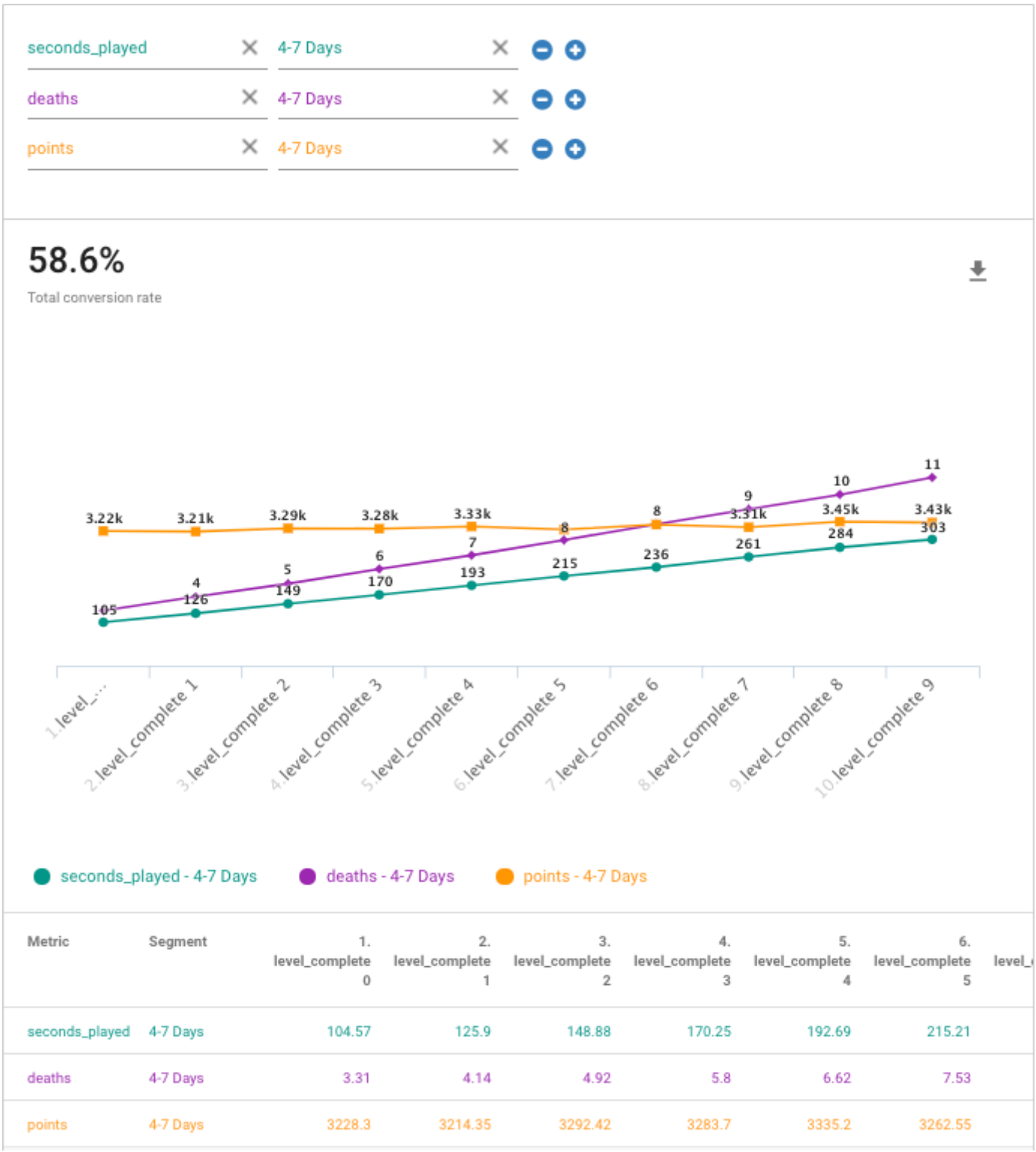


A funnel chart showing overall conversion from step to step

The above example shows an example funnel based on the level_complete event. This example exhibits large drop offs in the early levels. The chart can't tell you exactly what is wrong – the levels could be too hard, or they could be too easy – but it can reveal good places to look for game-play problems.

Funnel metrics

You can add metrics and segments to the chart to get more detail about what is going on in the funnel. The metrics available include parameters that you included with any of the Standard or Custom Events used by the funnel.

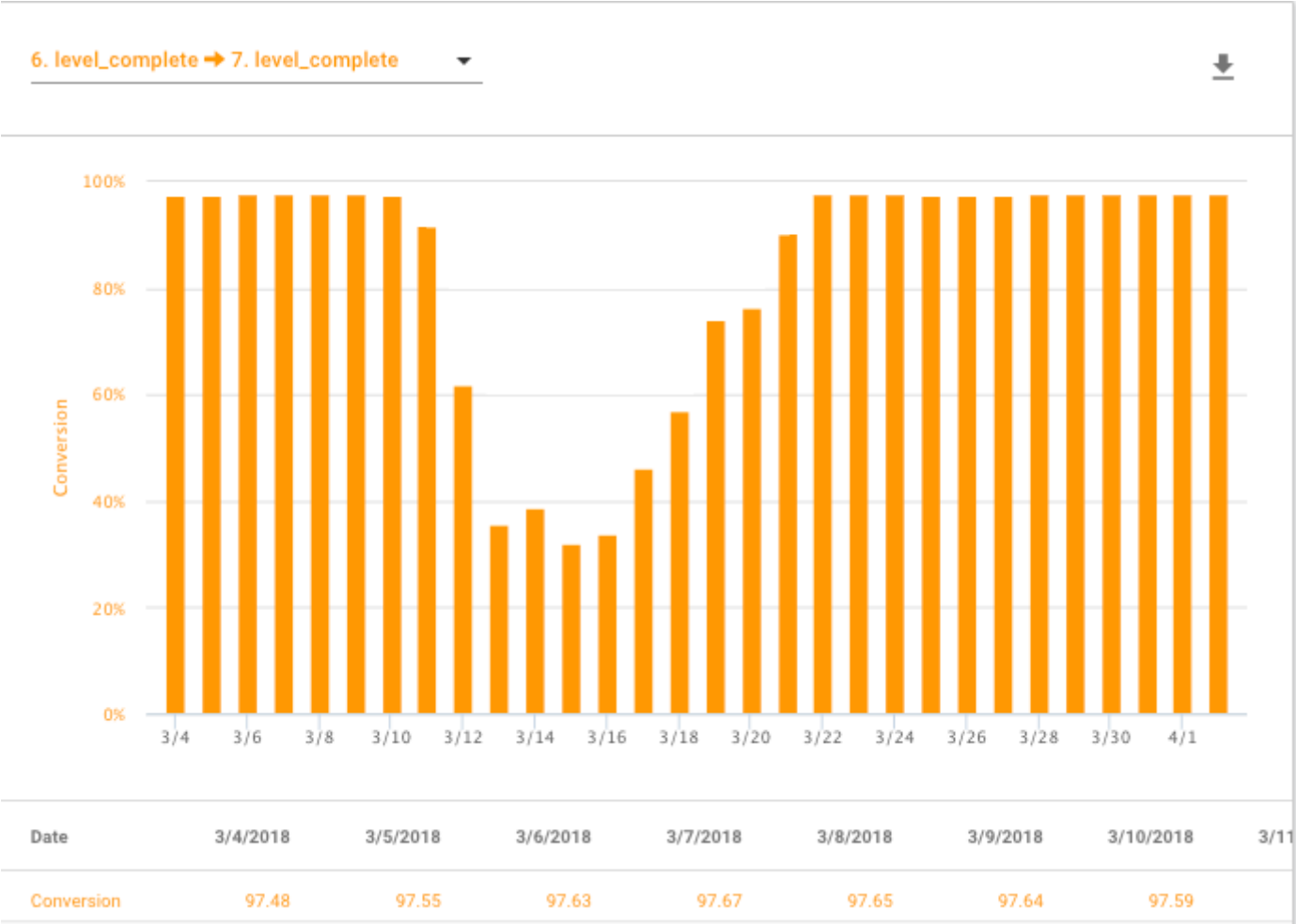


Metric	Segment	1. level_complete 0	2. level_complete 1	3. level_complete 2	4. level_complete 3	5. level_complete 4	6. level_complete 5	level_
seconds_played	4-7 Days	104.57	125.9	148.88	170.25	192.69	215.21	
deaths	4-7 Days	3.31	4.14	4.92	5.8	6.62	7.53	
points	4-7 Days	3228.3	3214.35	3292.42	3283.7	3335.2	3262.55	

A funnel chart showing level_complete event parameter values for players in the 1–3 Days segment

Funnel Steps By Date chart

The Funnel Steps By Date chart shows changes of the conversion rate over time. You can display either the overall conversion rate of players entering and completing the funnel, or the conversion rate between individual steps.



A Funnel Steps By Date chart showing conversion between two funnel steps

The above example shows a drop-off in conversion near the middle of the reporting period. The conversion rate recovers a few days later. Perhaps an update contained a bug that made it harder for players to complete the step and the bug was fixed in a subsequent update.

2017–08–29 Page published with editorial review

New feature in Unity 2017.1

Funnel Dashboard UI changed 2017–09–19

Remote Settings

Leave feedback

Use the Unity Analytics Service **Remote Settings** feature to change variables in your game or application directly from the **Analytics** dashboard. For example, if you are concerned that some levels in your game might be too hard or too easy, you can create settings that control the difficulty of your level bosses. Then, if your **Analytics** data shows that certain level bosses are too tough or too weak, you can tune the gameplay without releasing an update. You can even use segments to apply different settings to different groups of players.

Remote Settings can help you:

Adapt your game to different types of players
Tune the game difficulty curve in near real time
Roll out new features gradually while monitoring impact
Tailor game settings to different regions or other player segments
Turn seasonal, holiday, or other time-sensitive features on and off
Test your ideas and designs

After you make changes to your Remote Settings, every computer or device starting a new session of your application downloads the new configuration values. No update to the application is necessary. You can connect Remote Settings to individual **GameObject** properties, or you can read the key-value pairs directly in code and provide your own logic for handling setting changes.

This section describes creating and changing Remote Settings, and using Remote Settings in a Unity Project.

Find the Remote Settings page under the **Optimization** section of the **Unity Analytics** dashboard.

2017–08–28 Page amended with no editorial review

2017–08–28 - Service compatible with Unity 5.5 onwards at this date but version compatibility may be subject to change.

New feature in 2017.1

Added segmented Remote Settings: set different values for different segments in 2017.1

Creating and changing Remote Settings

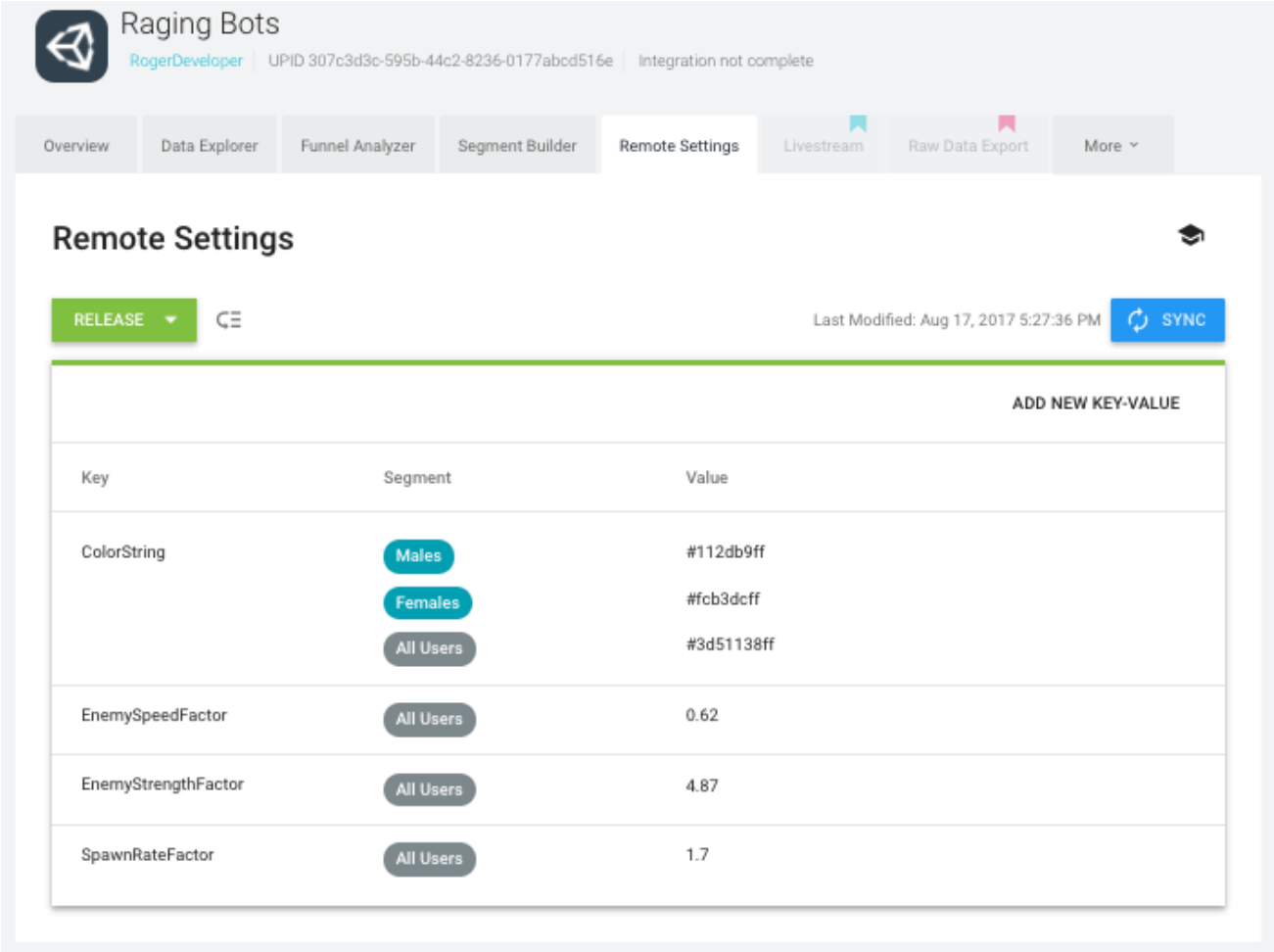
Leave feedback

Before using Remote Settings, you must first enable Unity Services and the Unity Analytics Service for your Unity project. Once you have enabled **Analytics**, open the **Analytics** dashboard and go to the **Remote Settings** page to create and change the Remote Settings values. See Using Remote Settings in a Unity project to learn how to access these Remote Settings values in your game or application.

The Analytics Service provides two Remote Settings configurations:

The Analytics Service sends Release configuration settings to computers and devices running regular, non-development builds of your application.

The Analytics Service sends Development configuration settings to computers and devices running **development builds** (that is, builds created with the Development Build flag checked on the Build Settings window). Play mode in the Unity Editor counts as a development build.



The Remote Settings page on the Analytics dashboard for a game called Raging Bots

Each Remote Setting consists of a key, a default value, and, optionally, segmented values. When a setting contains segmented values, the segments that the current player is in determine which values are included in the configuration. If more than one segment applies, the configuration includes the value from the highest priority segment.

You can create 200 values per configuration. A setting with one segment, uses one value; a setting with two segments uses two; and so on.

Setting names and value rules

An individual remote setting is a key-value pair. The rules for key names are:

Key names must be unique within the same configuration.

Key names must start with a letter.

Key names may contain only letters, digits, and the characters: ".", "_", and "-".

A setting value can be any of the following primitive types: **Int**, **Float**, **String**, or **Bool**. The rules for setting values are:

Int values are 32-bit integers (–2147483648 to 2147483647).

Float values are single-precision, 32-bit floating-point numbers (about -3.4×10^{38} to 3.4×10^{38}).

Strings are limited to 1024 characters.

You must specify boolean values with the strings "true" or "false".

Adding Remote Settings

To add a Remote Setting:

In your Analytics dashboard, open the **Remote Settings** page.

Set **Configuration** to **Release** or **Development** for the setting.

Click **ADD NEW KEY-VALUE** (at the top of the settings list).

Enter a name for the key.

Set the type for the value.

Enter the default value.

Click the **Save** button.

Click the **Sync** button to publish your changes. (You do not need to click the **Sync** button after creating each individual setting; you can wait to synchronize your settings until you finish making all changes to the current configuration.)

Note: To add values targeted to different segments, save the default key-value setting and then edit it. The segment options only appear when you edit a setting, not when you first create one.

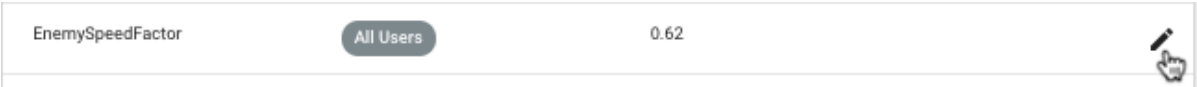
Editing Remote Settings

To edit a Remote Setting:

Open the **Remote Settings** page in your Analytics dashboard.

Choose the **Release** or **Development Configuration**.

Click the edit icon next to the setting you want to change.



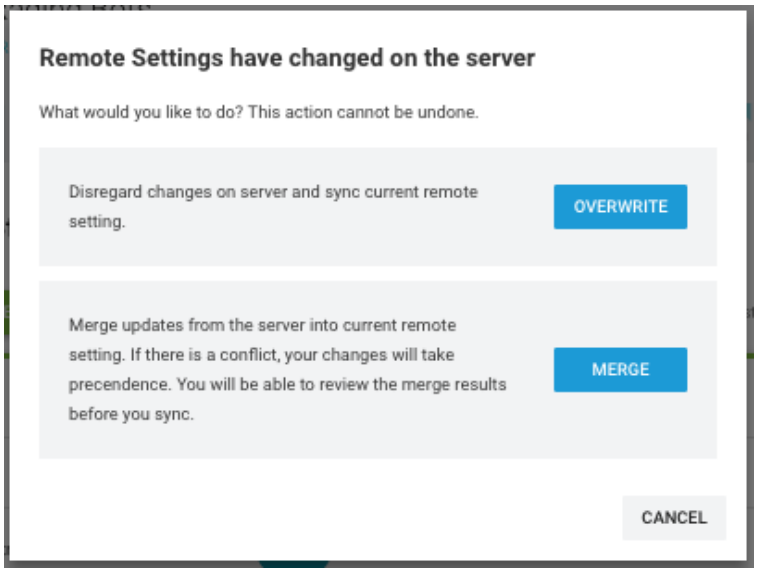
Make your changes. Note that changing a key name has the same effect as deleting the old key and creating a new one.

Click the **Save** button.

Click the **Sync** button to publish your changes. Unity reads the updated settings the next time players start a session.

Resolving simultaneous edits

If two people on your team edit the Remote Settings on a project at the same time, you can get a conflict when trying to synchronize the modified settings with the service.



Warning that someone else has changed the settings while you were editing the same project

When a conflict occurs, you have the following options:

OVERWRITE

Discards all changes on the service since you last synced and pushes your settings to the service. When you choose overwrite, the settings on the service will match your current settings exactly.

MERGE

Combines your current settings with the modified settings from the service and gives you the opportunity to make changes before completing the synchronization operation.

If you edit the same setting value, your changes overwrite the version from the service. In addition, values and settings deleted on the service are restored (since they still exist in your version). Otherwise, changes from the service since you last synced are retained.

After merging, your Remote Settings page is updated to reflect the merger, but the combined settings are not pushed to the server. You can review the updated changes and click the **Sync** button again when you are ready.

CANCEL

Cancels the synchronization operation and closes the conflict dialog without making any changes. Your settings are not saved to the service.

Adding different values for different segments

You can assign multiple segmented values to a single Remote Settings key. When a player is a member one of the segments for which a specific value exists, Unity sends that value to the player instead of the default, **All Users** value. (If a player qualifies for several segmented values, Unity sends the value from the highest priority segment.)

To set a value specific to a given segment:

Open the **Remote Settings** page in your Analytics dashboard.

Choose the **Release** or **Development** Configuration.

Add the Remote Setting if it does not already exist.

Click the edit icon at the far right of the setting.

Click **ADD EXISTING SEGMENT**.

In the new row, choose the segment from the drop-down list and enter a value.

Click the **Save** button.

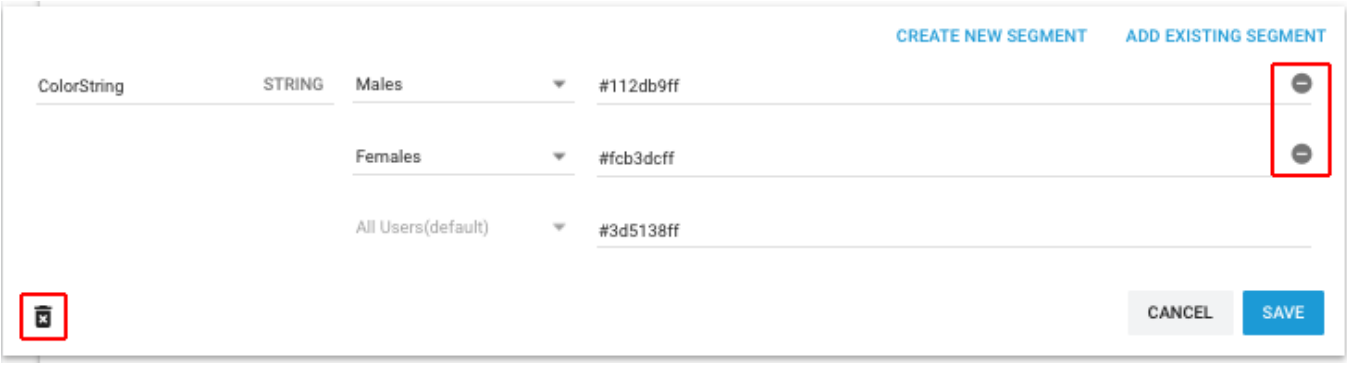
Deleting Remote Settings

To delete a Remote Setting, first open the edit view for the setting:

Open the **Remote Settings** page in your Analytics dashboard.

Set **Configuration** to **Release** or **Development** for the setting.

Click the edit icon next to the setting you want to delete to open the edit view:



To delete a value for a particular segment, click the **minus** button to the right of the edit view and then click **Save**. (You cannot delete the default **All Users** value.)

To delete the entire setting, click the delete icon at the bottom of the edit view and then confirm you want to delete the setting.

Click the **Sync** button to publish your changes.

Setting segment priority

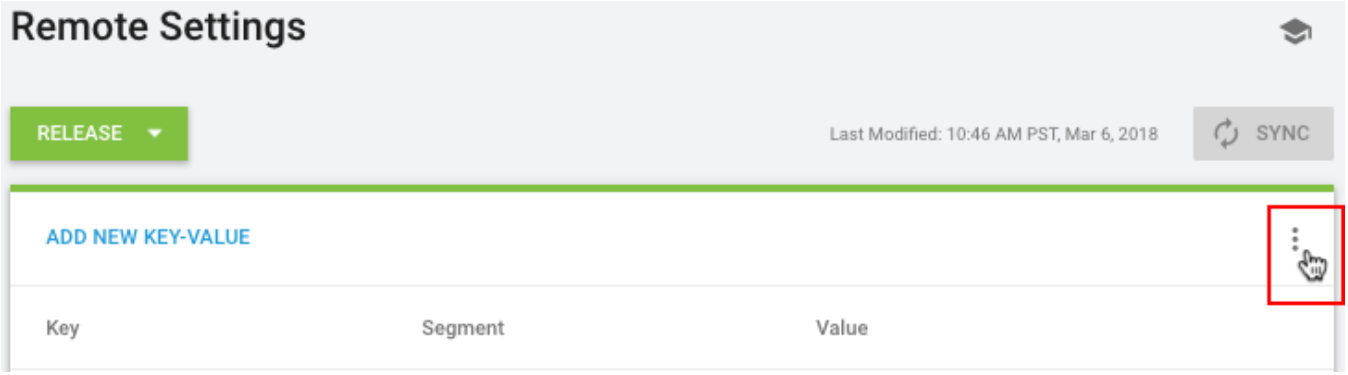
When a player is a member of more than one segment, Unity sends the value assigned for highest priority segment. (If a setting doesn't have any segmented values, or the player isn't a member of the segments that do have different values for a key, then Unity sends the All Users value.) You can set the order of your segments to specify which segmented values should take precedence when there is an overlap.

To set the segment priority:

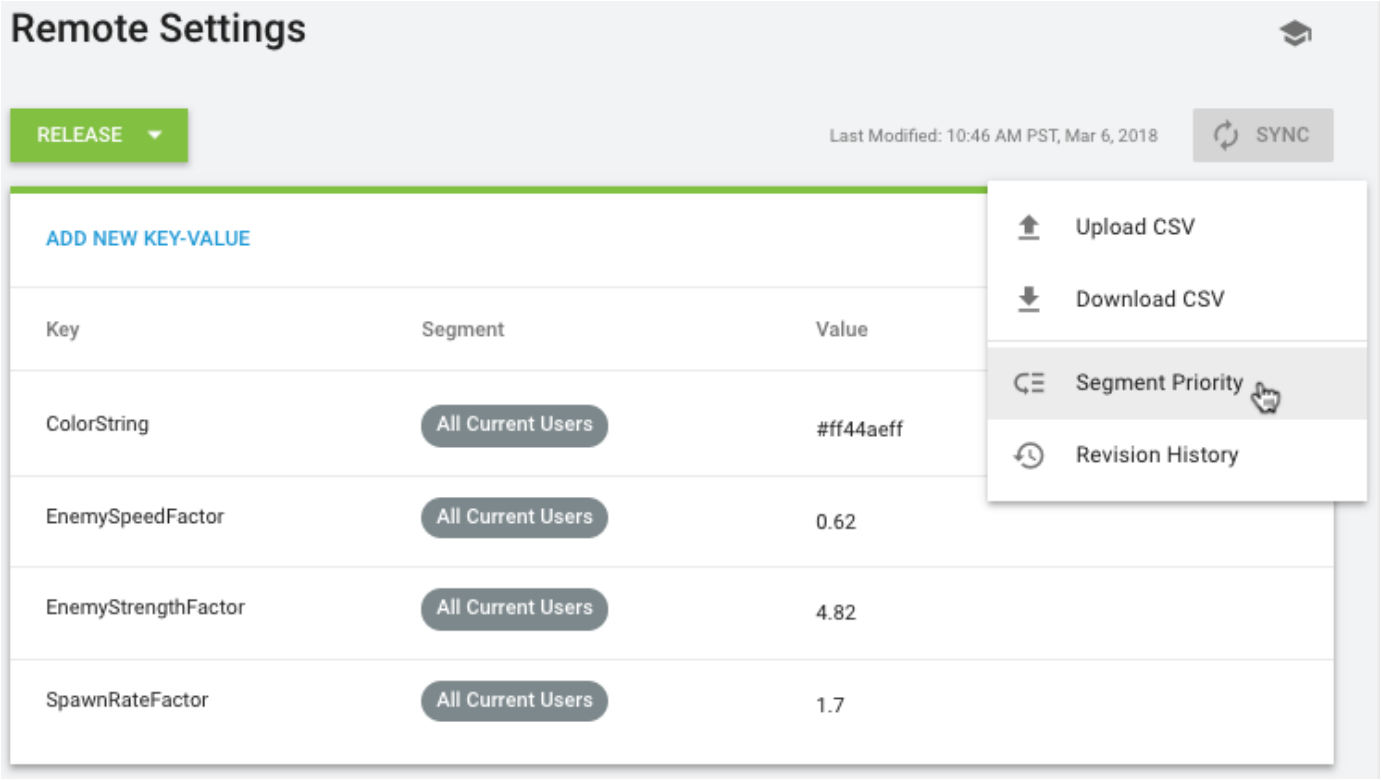
Open the **Remote Settings** page in your Analytics dashboard.

Choose the **Release** or **Development** Configuration.

Click the menu icon at the top, right corner of the settings list:

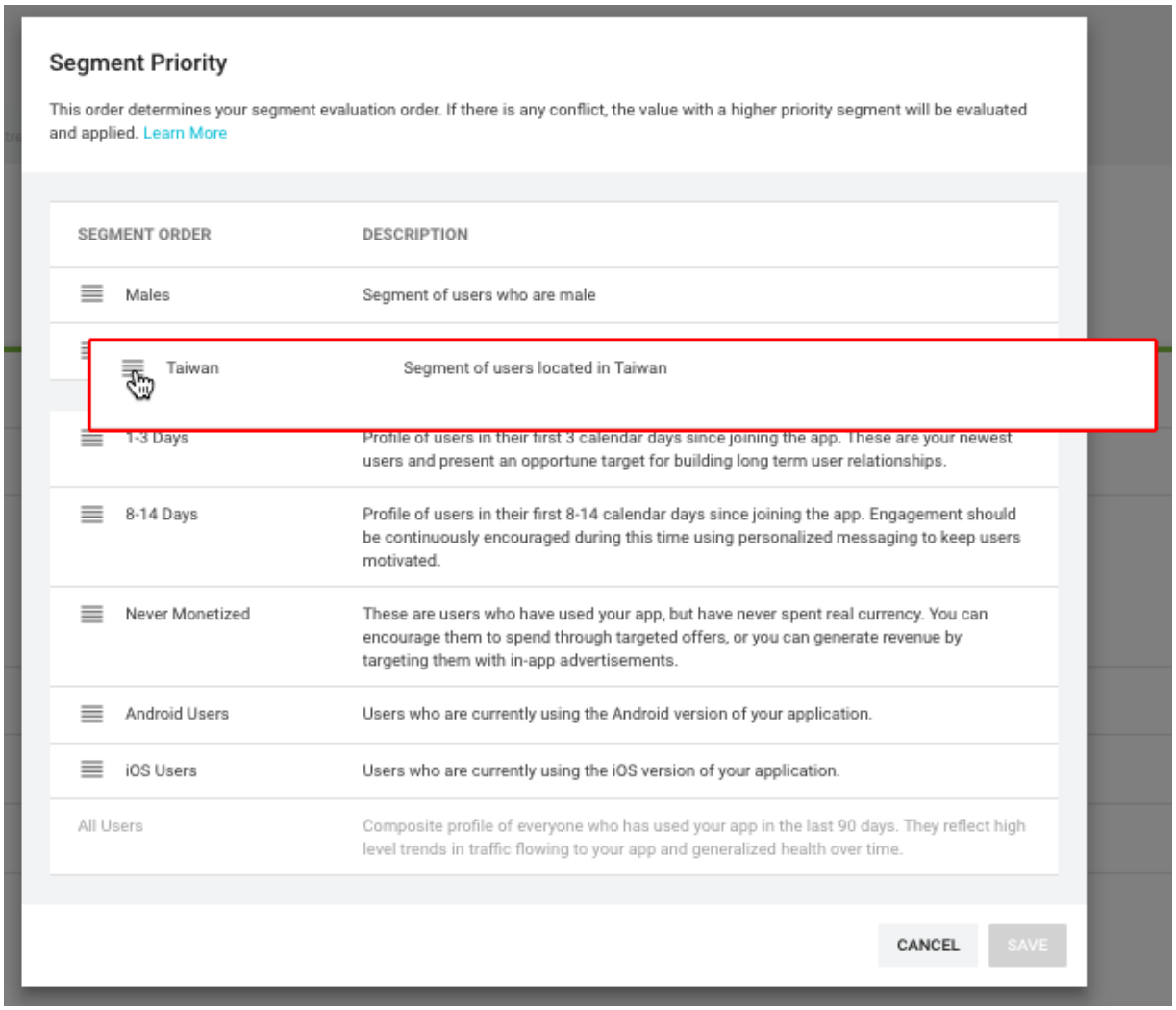


Click the segment priority icon.



The **Segment Priority** window opens, showing your current order. Only segments already assigned values for any Remote Settings appear in the window.

Reorder the segments by clicking and dragging the items into place.



When the order is correct, click **Save**.

Click the **Sync** button to publish your changes.

The segment priority applies to all settings within a configuration, but you can have a different priority for the Development and Release configurations.

Importing and Exporting Settings

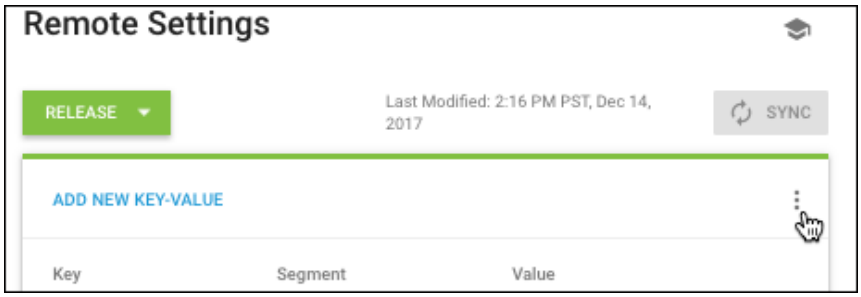
You can import and export settings using comma-separated-value (CSV) format files.

To export your current settings:

Navigate to the Remote Settings page on your **Unity Analytics** dashboard.

Choose the configuration, either **Release** or **Development** that you want to export.

Click the menu icon at the right side of the list header:



Choose **Download CSV** to export the current settings to your computer.

The Analytics Service downloads your current settings as a CSV file.

Note that if you download the CSV file before creating any setting, the file is completely empty. A valid file for uploading must contain the header row. See Remote Settings CSV Format.

To import settings into your project:

Navigate to the Remote Settings page on your Unity Analytics dashboard.

Choose the configuration, either **Release** or **Development** that you want to overwrite with the imported settings.

Click the menu icon at the right side of the list header.

Choose **Upload CSV** to import a new settings file to the project.

Choose the CSV file containing the new settings.

Click **Open**.

The Analytics Service uploads the settings file and replaces all the current settings in the selected configuration, but does not commit the values yet.

If a format or value error exists in the file, the service gives you an option to download a CSV file detailing the errors. No changes are made to the settings when an error occurs.

Click **Sync** to commit the setting changes and make them live. (Refresh the page or navigate away without synchronizing to discard any uploaded settings.)

When you import settings, **ALL** the keys and values are changed to match the uploaded file. If an existing setting or segment value does not appear in the file, that setting is deleted.

Creating and Deleting Settings during Import

In addition to editing setting values, you can create and delete settings when you upload a CSV file to for a configuration.

First, download the current setting CSV file for the configuration. You can then edit this file to make your changes. (See Working with Spreadsheets) if you use a Spreadsheet application to edit the file.)

To create a new setting:

Add a new row in the file.
Specify a unique key name.
Define the data type (int, float, string, or bool).
Enter “All Current Users” as the segment.
Leave the last field blank (for the default segment).
A new setting will look like:

```
a_bool_value,bool,TRUE,All Current Users,
```

To add a new segmented value for a setting:

Add a row to the file.

Repeat the key name and data type of the setting.

Enter the value for the segment.

Enter the segment name. The segment name must already be defined for the project before you import the settings. Use the Segment Builder page to view the existing segments as well as define new ones.

Set the segment priority.

The priorities for all segments in the file must be consecutive integers starting with 1. For example, if you use five segments in your CSV file, the only priorities that are allowed are the numbers 1–5. The same priority must be assigned wherever in the file that a segment is used. For example, if you use the Japan segment for a dozen values in your CSV file, the same priority value must be used in each row.

A setting with segmented values will look like:

```
a_bool_value,bool,false,All Current Users,
a_bool_value,bool,true,1-3 Days,1
a_bool_value,bool,false,4-7 Days,2
```

To delete a setting:

Simple delete the row defining the setting from the file. If you delete the row containing the default, “All Current Users” segment, you must also delete any rows defining segmented values. Note that if you delete segmented values you might also need to renumber the segment priorities to maintain the consecutive integer sequence.

Finally, uploaded the modified CSV file and press the **Sync** button on the Remote Settings page to save the changes and publish the changes.

Remote Settings CSV File Format

The Remote Settings CSV format conforms to ISO rfc4180. The exported file is encoded as UTF–8.

The settings file has five fields. You must include the header row as the first line in the file:

```
key,type,value,segment,priority
```

The fields in the file are:

key – The name of a Remote Settings key.
type – the setting data type: int, float, string, or bool.
value – the setting value.
segment – the name of the segment for a setting value.
priority – the priority of the segment for a setting value.

Rules for entries in the CSV file:

Each row must contain a unique combination of key and segment.

All segments in the CSV file must already exist. New segments must be created using the Segment Builder page of the Analytics dashboard.

Use the string, "All Current Users" for the default segment.

The default "All Current Users" segment must be associated with each key exactly once in the file.

Do not assign a priority to the "All Current Users" segment (leave the field blank).

Priorities for all other segments must be assigned.

A segment priority must be the same for all keys that segment is used in.

The segment priorities assigned across all segments must be consecutive integers (starting with 1).

Use TRUE and FALSE for Boolean values.

Strings containing commas, double quotes, and line breaks must be enclosed with double-quote characters.

Double quote characters in a string must be escaped by repeating the double quote. For example:

```
"Hello World!"            -> """Hello World!"""
He said, "How do you do?" -> "He said, ""How do you do?"""
```

The file can contain up to 201 rows (including header).

Tip: Create sample Remote Settings values on the dashboard and download to generate a sample file.

Working with Spreadsheets

The CSV file format is not rigorously standardized, so you may experience variations when opening or saving CSV files in different spreadsheet applications. Always test that the fields in your Remote Settings file are properly preserved after making the round trip from the Remote Settings page, to your editing application, and back. Especially check that Unicode characters (Emoji, symbols, characters outside the basic ASCII range, etc) are not altered and that floating point numbers are not truncated.

Usually when importing a CSV file, spreadsheet applications interpret strings starting with `'`, `+`, `-`, or `=` characters as expressions rather than plain strings. You can try enclosing such strings in double quotes and adjusting the import options of the spreadsheet you are using. For example, in Google Sheets you can turn off the "Convert strings to numbers and dates" option; while in LibreOffice you can turn on, "Quoted Strings As Text".

For settings of type float, make sure that all significant digits of the setting value are displayed in the spreadsheet cell. Otherwise, the spreadsheet application truncates the value and you lose precision when you export the spreadsheet to a CSV file.

2017–12–12 Page amended with no editorial review

2017–08–28 - Service compatible with Unity 5.5 onwards at this date but version compatibility may be subject to change.

2017–06–30 - Added: Allow the characters “.”, “_”, and “-” in key names.

2017–06–30 - Added: CSV import and export.

New feature in 2017.1

2017–08–28 - Added segmented Remote Settings: set different values for different segments in 2017.1

Using Remote Settings in a Unity project

Leave feedback

To use remote settings in a project, first install the Remote Settings package from the Unity Asset Store. These **Assets** provide support for viewing your Remote Settings in the Unity Editor, and for mapping Remote Settings values to the fields and properties of **components** on your **GameObjects**. See Creating and changing Remote Settings to learn how to define your settings with the Unity **Analytics** Dashboard.

You can use the Remote Settings component to connect remote settings to the appropriate fields and properties of your GameObjects.

You can also use the RemoteSettings class to write your own script logic for handling Remote Settings.

This section describes how to use Remote Settings in your games and applications:

Remote Settings network requests

Enabling Remote Settings

Managing Remote Settings in the Unity Editor

Remote Settings component

Remote Settings scripting

Testing Remote Settings

2017–05–30 Page published with editorial review

2017–05–30 - Service compatible with Unity 5.5 onwards at this date but version compatibility may be subject to change.

New feature in 2017.1

Remote Settings network requests

Leave feedback

When you create a Remote Settings key-value pair in the Unity **Analytics** Dashboard, the Unity **Analytics** Service stores that setting in the **Configuration** for your project that you have specified (either the **Release** or the **Development** configuration). Whenever a player starts a new session of your application, Unity makes a network request for the latest configuration from the Analytics Service. Unity considers a new session to have started when the player launches the application, or returns to an application that has been in the background for at least 30 minutes. Unity requests the **Release** configuration when running regular, non-development builds of your application, and requests the **Development** configuration when running development builds. Play mode in the Unity Editor counts as a development build.

Note: For Unity to request the **Development** configuration, you must build the application with Unity version 5.6.0p4+, 5.6.1p1+, 2017.1+, or Unity 5.5.3p4+, and tick the **Development Build** checkbox on the Build Settings window. If you build the game with an older version of Unity, Unity always requests the **Release** configuration.

When the network request for the Remote Settings configuration is complete, the RemoteSettings object dispatches an `Updated` event to any registered event handlers, including those registered by Remote Settings components.

If the computer or device has no Internet connection and cannot communicate with the Analytics Service, Unity uses the last configuration it received and saved. The `RemoteSettings` object still dispatches an `Updated` event when using a saved configuration. However, if Unity has not saved the settings yet (such as when a player has no network connection the first time they run your game), then the `RemoteSettings` object does not dispatch an `Updated` event, and so does not update your game variables. Requesting the Remote Settings configuration over the network is an asynchronous process that might not complete before your initial **Scene** has finished loading, or might not complete at all, so you should always initialize your game variables to reasonable defaults.

Note: The web service from which Unity downloads the Remote Settings configuration is read-only, but is not secured. This means that the configuration could be read by third-parties. You should not put sensitive or secret information into your Remote Settings. Similarly, the saved settings file could be read and modified by end-users (although any modifications are overwritten the next time a session starts with an available Internet connection).

2017–05–30 Page published with editorial review

2017–05–30 - Service compatible with Unity 5.5 onwards at this date but version compatibility may be subject to change.

New feature in 2017.1

Enabling Remote Settings

Leave feedback

To enable Remote Settings in a project:

If you haven't already, turn on Unity Analytics for the project.

If needed, import the Remote Settings package from the Unity Asset Store.

Open the Remote Settings window with **Window** > **Unity Analytics** > **Remote Settings**:

Enter the **Project Secret Key** for your **Analytics** project and click **Next**. You can find your **Project Secret Key** under the **Configure** page on the Unity Analytics dashboard

On the Remote Settings window, click the **Refresh** button. The window displays the key-value pairs for any settings you have already created.

If you haven't created any Remote Settings yet, see Creating and changing Remote Settings to get started.

Note: Keep your **Project Secret Key** secret. Anyone possessing this value could access your Analytics data. The secret keys for your projects are stored in your Editor preferences file, not in project files. The **Project Secret Key** was formerly known as the **Raw Data Export API Key**. The key value is the same; use the **Project Secret Key** in any place where you previously used the **Raw Data Export API Key**.

2017–06–30 Page published with editorial review

2017–05–30 - Service compatible with Unity 5.5 onwards at this date but version compatibility may be subject to change.

New feature in 2017.1

Managing Remote Settings in the Unity Editor

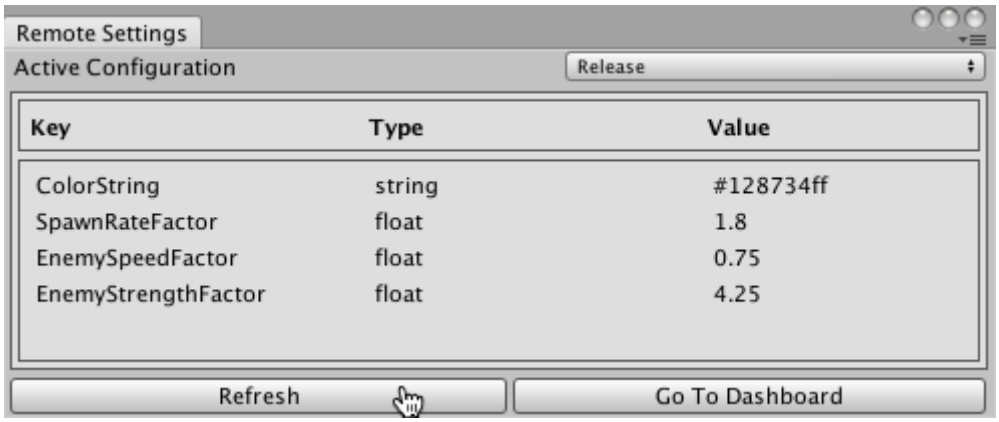
Leave feedback

The Remote Settings window helps you manage your remote settings while you are developing your project in the Editor. (Use the Unity Analytics Dashboard to create and edit the settings themselves.)

The Remote Settings window is not included in the standard Unity download and install. It is part of the Remote Settings **Asset** package, which is a Unity plug-in. Download the Remote Settings **Asset** package from the Unity Asset Store and import it into your project.

To open the Remote Settings window, go to **Window** > **Unity Analytics** > **Remote Settings** in the Unity Editor. In order for the Editor to fetch your Remote Settings from the Analytics Service, you must first supply the Project Secret Key as described in Enabling Remote Settings.

The Remote Settings window, part of the Remote Settings plug-in, displays the key-value pairs for the settings defined on your **Analytics** Dashboard.



Remote Settings window

Click **Refresh** to fetch the latest remote settings. The Editor also fetches the most recently synchronized settings when you enter Play mode.

Set **Active Configuration** to **Release** or **Development** to choose which set of key-value pairs to work with in the Editor. Note that the **Development** configuration is always used in Play mode in the Editor. To test the **Release** configuration, use **File** > **Build and Run**, and make sure that the **Development Build** box is unchecked on the Build Settings window.

2017–05–30 Page published with editorial review

2017–05–30 - Service compatible with Unity 5.5 onwards at this date but version compatibility may be subject to change.

New feature in 2017.1

Remote Settings component

Leave feedback

Use the Remote Settings component to control the properties of other components in your **Scene** without writing any code. The Remote Settings component is part of the Remote Settings plug-in, which you can download from the Unity Asset Store.

Before using the Remote Settings component, you must enable Remote Settings in your project and should also create your Remote Settings key-value pairs using the Unity Analytics Dashboard.

You can place the Remote Settings component on the same **GameObject** as another component you want to control, or place it on a different **GameObject**. The only requirement is that both the Remote Settings component and any controlled components are active in the same Scene.

Using the Remote Settings component

To connect a remote setting to a component property or field:

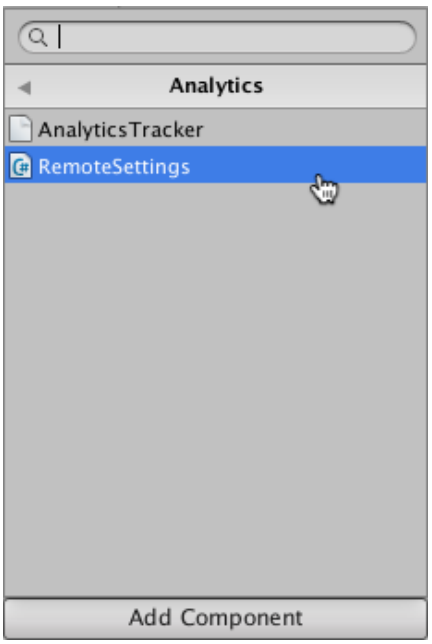
Go to **Window** > **Unity Analytics** > **Remote Settings** to open the Remote Settings window.

Choose the configuration (**Release** or **Development**) that contains the setting you are going to connect.

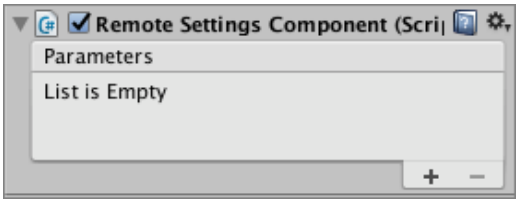
Go to the **Inspector** window for the GameObject that you want to hold the Remote Settings component.

Click the **Add Component** button.

Find the **Analytics** > **RemoteSettings** script in the list.

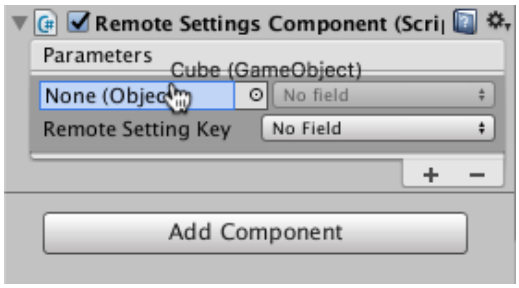


Click **Add Component** to add the Remote Settings component to the GameObject.



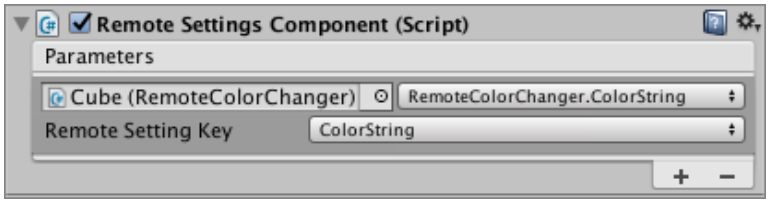
To add a new parameter mapping, click the **+** icon at the bottom of the Remote Setting component's **Parameters** list.

Drag the GameObject or the component that you want to remotely control to the parameter's **Object** field.



Select the property or field you want to control in the parameter's **Field** drop-down list.

Choose the **Remote Setting Key** that you want to use to control this component property or field.



Click the **+** icon to add additional parameters.

If no **Remote Settings Key** names appear in the list, open the **Remote Settings** window (menu: **Window** > **Unity Analytics** > **Remote Settings**) and click **Refresh**. If your remote settings still do not display in this window, check that you have an Internet connection and that your project is properly set up (see Enabling Remote Settings).

If the wrong keys appear in the list of key names, open the Remote Settings window (menu: **Window** > **Unity Analytics** > **Remote Settings**) and set the **Active Configuration** to the configuration containing the correct set of keys.

The Remote Settings component cannot set the variables of prefabs that Unity loads later in the Scene than the component itself. Similarly, a Remote Settings component that is loaded later in the Scene can only set variables of objects that are part of the same prefab. Use more than one Remote Settings component to handle these types of situations.

Setting non-primitive properties

You can use the Remote Settings component to set the primitive fields and properties of an object directly. However, to set the variables of an object's non-primitive members, you do have to write some additional code. The easiest approach is to add primitive-type properties to an object that you can set using the Remote Settings component. Then, implement the setter functions of these properties to update the intended variables of the non-primitive objects.

Code example

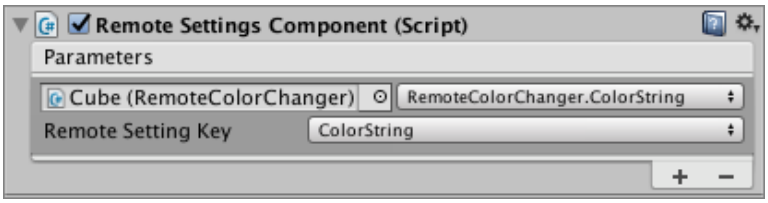
In the example below, the class sets the base color of the Material assigned to a rendered GameObject. To do this, the class defines a primitive string-type property that takes an HTML-style color string. The setter for this property parses the string and sets the Material color accordingly.

```
using UnityEngine;

public class RemoteColorChanger : MonoBehaviour
{
    private string _colorString = "";
    public string ColorString {

        get {
            return _colorString;
```

```
        }

        set {
            Color colorObject;
            if (ColorUtility.TryParseHtmlString (value, out colorObject)) {
                _colorString = value;
                Renderer renderer = GetComponent<Renderer> ();

                if (renderer != null) {
                    MaterialPropertyBlock materialProperties = new MaterialPropertyBloc
                    renderer.GetPropertyBlock (materialProperties);
                    materialProperties.SetColor ("_Color", colorObject);
                    renderer.SetPropertyBlock (materialProperties);
                }
            } else {
                Debug.LogWarning ("Invalid color string: " + value);
            }
        }
    }
}
```

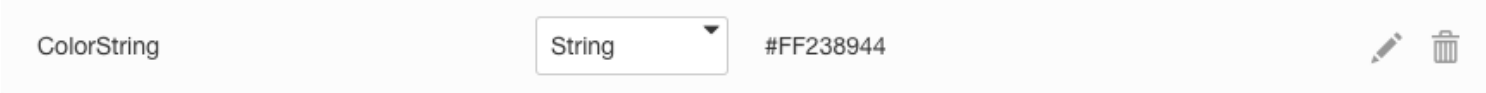
Using the code example

You can add this `RemoteColorChanger` script to any `GameObject` that has a `Renderer` component. You can then use the Remote Settings component to map a setting key to the `ColorString` property. In this example, the script is a component of a `Cube` object.



A Remote Settings component mapping the `ColorString` Remote Setting key

The matching key-value pair on the Remote Settings page of your Analytics Dashboard looks like the following:



The `ColorString` setting defined on the Analytics Dashboard

Use the same technique to set any non-primitive value.

2017–05–30 Page published with editorial review

2017–05–30 - Service compatible with Unity 5.5 onwards at this date but version compatibility may be subject to change.

New feature in Unity 2017.1

Remote Settings scripting

Leave feedback

Use the Unity Scripting API RemoteSettings class to access your remote settings in code. The `RemoteSettings` class provides functions to access your remote settings, like `RemoteSettings.GetString()`, as well as events that you can use to trigger functions when your settings are fetched or updated.

Event	Purpose
BeforeFetchFromServer	Dispatched immediately before the `RemoteSettings` class makes its network request to retrieve your remote settings.
Completed	Dispatched when the network request is complete. `Completed` is always dispatched, even when the network request fails and when there are no changes to the remote settings. You can use the `serverResponse` code passed to your delegate function to determine whether the request succeeded or failed. The parameter contains one of the standard HTTP response codes, such as 200 for success.
Updated	Dispatched after the network request completes successfully, but ONLY when your remote settings have changed since the previous check. Note: Prior to Unity 2018.1, the `Updated` event was always dispatched, even when there were no settings changes. If you rely on a delegate function to apply settings in your game whether or not there have been changes, you must use the `Completed` event rather than the `Updated` event.

You should register your handler functions for these events as early as possible during application launch, such as in the `Awake()` function of a MonoBehaviour loaded in your first Unity **scene**.

Because fetching the settings requires a network transaction, the `RemoteSettings` object dispatches its events asynchronously. Your handler functions might not be called in the same order relative to other events on every platform or even on every launch of the same platform. Always initialize your configuration variables with reasonable default values, and allow for the possibility that your event handlers can be called at different times during application startup.

Code example

The following example shows a class that defines a number of properties for tuning game difficulty and updates these properties based on remote settings:

```
using UnityEngine;

public class RemoteTuningVariables : MonoBehaviour {

    public float DefaultSpawnRateFactor = 1.0f;
    public float DefaultEnemySpeedFactor = 1.0f;
    public float DefaultEnemyStrengthFactor = 1.0f;
    public static float SpawnRateFactor{ get; private set; }
```

```
    public static float EnemySpeedFactor{ get; private set; }
    public static float EnemyStrengthFactor{ get; private set; }

    void Start () {
        SpawnRateFactor = DefaultSpawnRateFactor;
        EnemySpeedFactor = DefaultEnemySpeedFactor;
        EnemyStrengthFactor = DefaultEnemyStrengthFactor;

        RemoteSettings.Completed += HandleRemoteSettings;
    }

    private void HandleRemoteSettings(bool wasUpdatedFromServer, bool settingsCh
        SpawnRateFactor
            = RemoteSettings.GetFloat ("SpawnRateFactor", DefaultSpawnRateFactor
        EnemySpeedFactor
            = RemoteSettings.GetFloat ("EnemySpeedFactor", DefaultEnemySpeedFact
        EnemyStrengthFactor
            = RemoteSettings.GetFloat ("EnemyStrengthFactor", DefaultEnemyStreng
    }
}
```

Notice that the class provides default values in the `RemoteSettings.GetFloat()` method calls. If the `RemoteSettings` object cannot find the specified key (if you misspell a key name, for example), then the method still assigns your default values to the tuning variables. Otherwise, the `GetFloat()` and `GetInt()` methods assign zero to numbers, `GetString()` assigns an empty string to strings, and `GetBool()` assigns false to boolean variables.

The class also assigns the same default values to the properties in the `Start()` method in case another class accesses the settings before the remote settings are fetched and no previously cached configuration is available locally. Assigning the default values in the `Start()` method ensures that the properties always have reasonable values.

Ensuring setting consistency

If you have several classes in your game that access remote settings at different times, it is possible that some objects will access your settings before the asynchronous network request completes and some will access the settings afterwards, possibly resulting in inconsistent setting values. To avoid this inconsistency problem, you either have to wait for the asynchronous network request to complete before accessing any settings or you can simply use the existing settings you have at the start of the current session. In this latter case, the local settings are updated in the background and a player will see them in the next session.

Method	Benefits	Drawbacks

Method	Benefits	Drawbacks
Wait for request completion	- Setting values are consistent - Always using latest settings	- Indeterminate delay in setting availability - Asynchronous code complexity
Use current settings	- Setting values are consistent - Settings are available immediately	- New setting values not used until the second session - The first time a player runs the game, no settings are available. (Use default values.)

The following example illustrates a provider class that assigns its properties based on the last version of your remote settings fetched from the **Analytics** service. The `RemoteSettings` class still updates the settings in the background, so any settings changes are used in the next session (rather than the current session). The following example uses a singleton pattern so that all parts of your game that use these settings get the same values no matter when the `RemoteSettings` class receives the remote update.

```
using UnityEngine;

public class RemoteSettingProvider : MonoBehaviour {
    public float DefaultSpawnRateFactor = 1.0f;
    public float DefaultEnemySpeedFactor = 1.0f;
    public float DefaultEnemyStrengthFactor = 1.0f;
    public float SpawnRateFactor{ get; private set; }
    public float EnemySpeedFactor{ get; private set; }
    public float EnemyStrengthFactor{ get; private set; }

    // Singleton pattern
    private static RemoteSettingProvider _instance;
    public static RemoteSettingProvider Instance
    {
        get { return _instance; }
    }

    void Awake()
    {
        if (_instance != null && _instance != this)
        {
            Destroy(this.gameObject);
            return;
        }

        _instance = this;
        DontDestroyOnLoad(this.gameObject);

        //Use the current settings, before remote update, or default values if
        //the settings don't exist (yet)
```

```
        SpawnRateFactor
            = RemoteSettings.GetFloat("SpawnRateFactor", DefaultSpawnRateFactor)
        EnemySpeedFactor
            = RemoteSettings.GetFloat("EnemySpeedFactor", DefaultEnemySpeedFacto
        EnemyStrengthFactor
            = RemoteSettings.GetFloat("EnemyStrengthFactor", DefaultEnemyStrengt
    }
}
```

Using this method, the default settings are always used the first time a player runs your game after installation.

You can also use a hybrid approach that checks whether local settings exist using `RemoteSettings.GetCount()`. If the settings exist locally, you can use them immediately, but if not then you add an event handler to wait for the settings `Updated` event:

```
SpawnRateFactor
    = RemoteSettings.GetFloat("SpawnRateFactor", DefaultSpawnRateFactor);
EnemySpeedFactor
    = RemoteSettings.GetFloat("EnemySpeedFactor", DefaultEnemySpeedFactor);
EnemyStrengthFactor
    = RemoteSettings.GetFloat("EnemyStrengthFactor", DefaultEnemyStrengthFactor)


if(RemoteSettings.GetCount() == 0){
    //If there are no settings, get them from remote
    RemoteSettings.Updated += HandleRemoteSettings;
    Ready = false;
} else {
    Ready = true;
}
```

Other classes that use your remote settings must check the `Ready` property of your provider class to determine whether it is safe to access the settings. The following example illustrates a class that waits for the settings and also uses a configurable timeout:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class RemoteSettingConsumer : MonoBehaviour {
```

```
    public float SpeedFactor = 0;
    public float StrengthFactor = 0;
    public float SpawnRateFactor = 0;
    public float Timeout = 0.125f;

    void Start () {
        StartCoroutine(GetRemoteSettings(Time.time));
    }

    IEnumerator GetRemoteSettings(float startTime){

        while(!RemoteSettingProvider.Instance.Ready && (Time.time - startTime) <
            yield return null;

        SpeedFactor = RemoteSettingProvider.Instance.EnemySpeedFactor;
        StrengthFactor = RemoteSettingProvider.Instance.EnemyStrengthFactor;
        SpawnRateFactor = RemoteSettingProvider.Instance.SpawnRateFactor;
    }
}
```

If the timeout elapses before the remote settings become available, then your default values are used. Using a timeout allows the game to get the latest settings in the normal case where the network transaction completes in a very short amount of time, while not delaying startup unduly by waiting for the request itself to timeout in the case of network failure.

2018–08–23 Page published with editorial review

New feature in 2017.1

2018–08–23 - Updated event no longer dispatched every session as of 2018.1.

2018–08–23 - Service compatible with Unity 5.5 onwards at this date but version compatibility may be subject to change.

Testing Remote Settings

Leave feedback

While you can view the keys and values of both configurations in the Editor, Unity only loads one configuration at runtime. Unity uses the Development configuration in Play mode and in any builds created with the **Development Build** box checked on the Build Settings window. Unity uses the **Release** configuration in non-development builds.

Before testing Remote Settings, you must first create the settings and then either use the Remote Settings component or write the code necessary to map the settings to variables in your game or application.

To test with the **Development** configuration, click the **Play** button in the Editor or:

Go to **File** > **Build Settings** to open the Build Settings window.

Check the **Development Build** box.

Click the **Build And Run** button.

To test with the **Release** configuration:

Go to **File** > **Build Settings** to open the Build Settings window.

Uncheck the **Development Build** box.

Click the **Build And Run** button.

Clearing Local Settings

To test whether your game behaves as expected on first run when no local settings are present, you can clear the Application.persistentDataPath directory. If you are testing in the Unity Editor, you must exit from the Editor before clearing the directory.

2017–05–30 Page published with editorial review

2018–08–13 - Service compatible with Unity 5.5 onwards at this date but version compatibility may be subject to change.

New feature in 2017.1

Unity Analytics A/B Testing

Leave feedback

An A/B test is a controlled experiment that exposes different groups of players to variations of some aspect of your game. The aspect of a game that you test can be fundamental, such as the input control scheme, or it can be cosmetic, like a different color treatment, or something else in between. An A/B test allows you to compare key metrics collected during the experiment to help you analyze which variation performs better.

How does A/B testing work?

Unity **Analytics** A/B testing groups players according to the variants you define in your experiment. Each group sees the treatment of your game that you specify for that variant group. A treatment is a set of one or more Remote Settings to which you assign unique values for each variant group. At runtime, the A/B testing service randomly assigns a player to one of the groups based on your variant allocation percentages. The player receives the values of the Remote Settings for their assigned group. Your application then uses those Remote Settings so that the player experiences the aspect of your game you wish to test.

The key parts of an A/B test include:

Experiment — a test run by showing different treatments to different groups of players.

Variant — defines the number of groups and the distribution of players to those groups. One variant is always the control group.

Treatment — defines the Remote Settings to use for the experiment and the specific setting values for each variant.

Goal — The Analytics metric that you want to improve with the experiment. An experiment compares the goal metric across the variant groups so that you can see which treatment has the most desirable effect.

You can run more than one experiment at the same time, but you should be careful that the different experiments do not interact since some players will see the experimental treatments for both experiments. For example, if you run a test to compare the ordering of the steps in your onboarding tutorial in one experiment and run a different test to test tutorial pacing, you will have difficulty figuring out which experiment caused any changes in results. While that example might be pretty obvious, even tests of unrelated parts of your game could interact. The more fundamental the part of your game you are testing and the larger your experimental groups, the more chance there is for experimental interaction.

Create and run A/B tests from the **Unity Analytics** Dashboard. You must also implement the experiment treatment in your application using the Remote Settings API.

Creating an experiment

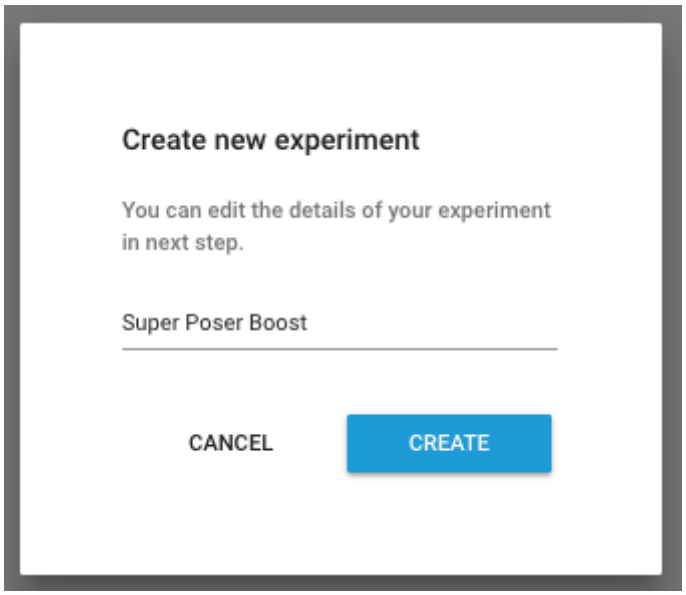
To create an experiment:

Go to the Analytics Dashboard for your project.

Click **A/B Testing** to open the A/B Testing page.

Click the **Create Experiment** button.

The **Create new experiment** dialog opens.



Enter a name for the experiment.

Click **Create**.

Once the experiment has been created, you must define the goals, variants and the treatments.

Choose an experiment goal

The experiment goal defines the analytics metric that the experiment treatments are designed to influence. You can choose:

Day 1 Retention — the percentage of users who return to your game exactly one day after the first time they play.

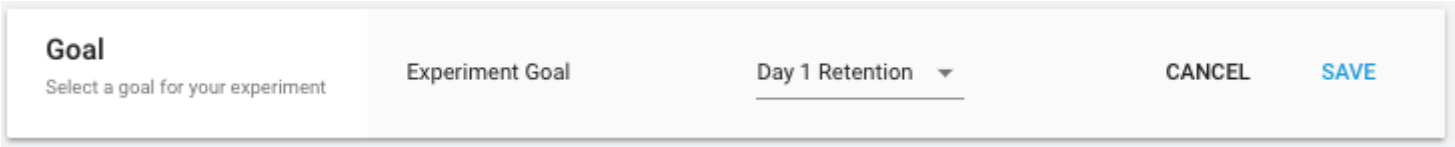
Day 7 Retention — the percentage of users who return to your game exactly seven days after the first time they play.

The report generated for your experiment shows the difference in the metric across the experiment variants. The chosen goal also influences the amount of time an experiment must run to collect statistically significant data. For example, it takes longer to run an experiment measuring Day 7 retention than one measuring Day 1 retention.

To choose the experiment goal:

Click the **Select Experiment Goal** button.

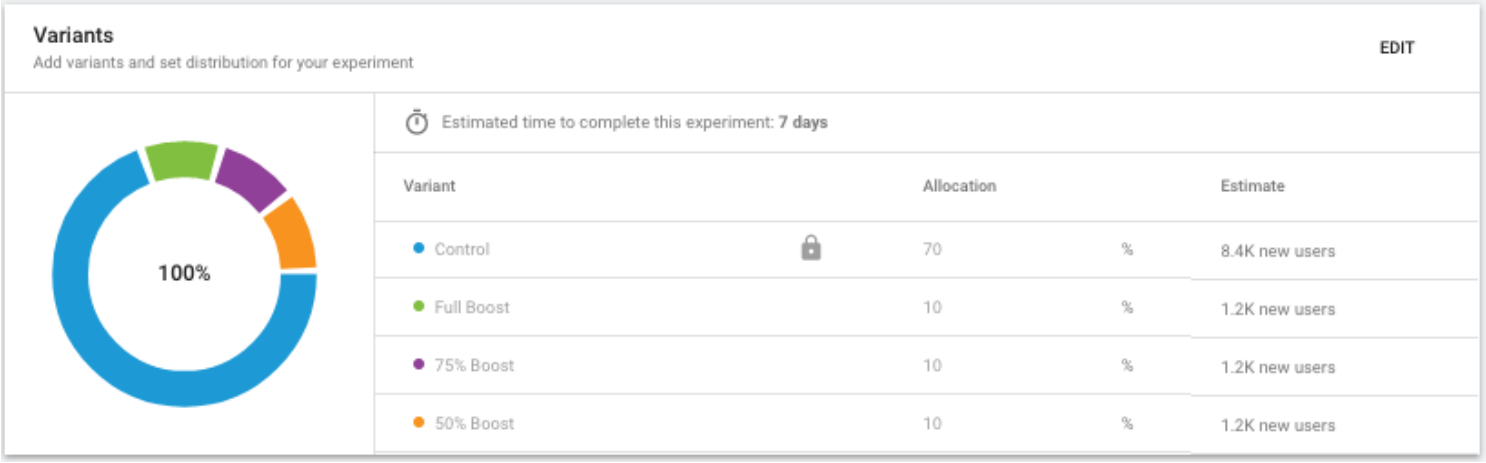
Choose a goal from the drop-down list.



Click **Save**.

Set up the experiment variants

Use the **Variants** section of the experiment to define the player groups for the A/B test and the assign the percentage of players the testing service allocates to each group.



Variants section

A **Control** group always exists. The control group should receive the current or default experience in your game. This group provides the basis to which the results of the other groups can be compared.

To add an additional variant, click the **ADD** button at the top of the section. Assign a name and an allocation percentage for the group. The allocation percentages for all groups must add up to 100%. The **Estimate** column displays the number of players expected to be allocated to each variant per day.

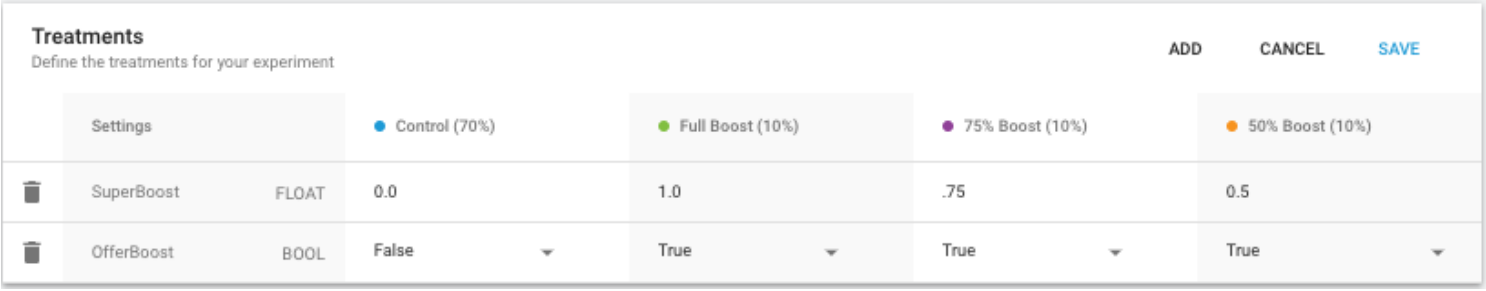
To delete a variant, click the trash can icon next to its name.

The estimated time to complete the experiment is based on the on how long it should take to achieve a statistically valid number of samples. The more players (DAU) your application typically has and the larger the allocation of players to the experimental variants, the shorter the estimated time will be.

Click **Save** when your variants are set up as desired.

Set up the experiment treatments

Use the **Treatments** section to define the settings used in the experiment and assign the individual setting values for each group.



Treatments section

Click the **ADD** button to add a setting to the experiment.

You can use existing Remote Settings from the Release configuration as long as the settings are not used in any other active experiments. Note that the Remote Settings page only displays the default value used for the control group. The variant settings can only be viewed or changed on the A/B Testing page.

You can create new settings without going to the Remote Settings page. Settings created in A/B Testing are not shown on the Remote Settings page. If you want to continue using a setting after the experiment is finished, you can create a Remote Setting with the same name.

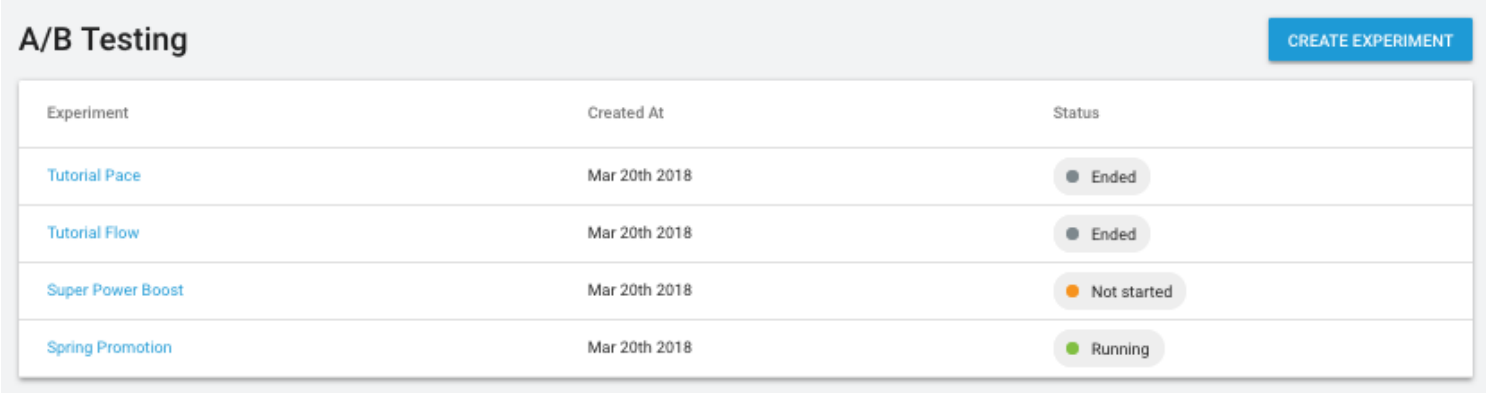
Click **Save** when your treatments are defined.

Implementing your treatments

Use the Remote Settings API to access the values defined in your experiment at runtime. Use the setting name shown in the Treatments section as the key when accessing the setting in your game. You can implement Remote Settings in code or use the Remote Settings component to access and apply the settings values in the Unity **Inspector** window.

Managing experiments

After you create at least one experiment, the A/B Testing page displays your experiments in a list.



Experiment queue

To start an experiment:

Click the name of an experiment in the **Not Started** status.

Click the **Start Experiment** button.

Note: After an experiment has started, you can no longer make changes, since that would invalidate the test results.

To stop an experiment:

Click the name of an experiment in the **Running** status.

Click the **Stop Experiment** button.

Note: You cannot restart an experiment.

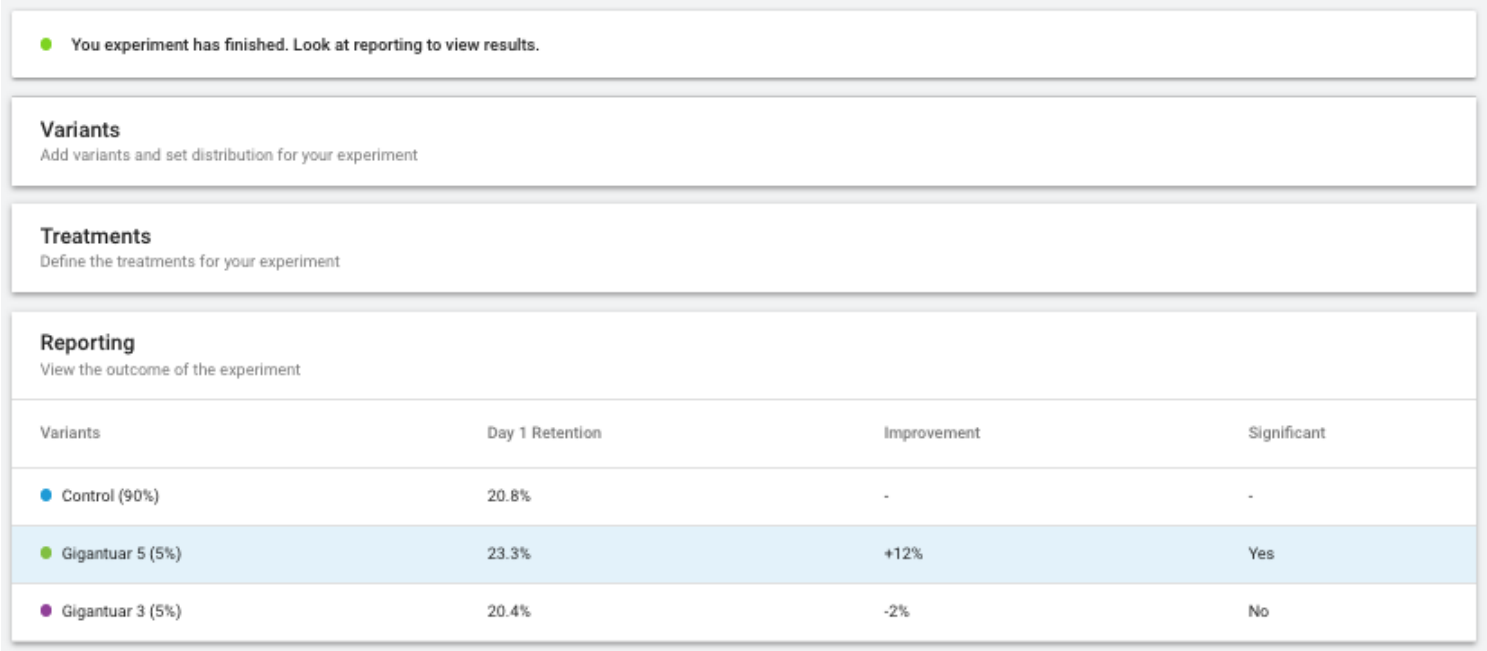
To view a report:

Click the name of an experiment in the **Ended** status.

The Reporting section displays the experiment results.

Reports

Use the Reports section to review the results of an experiment once it has finished.



Reports section

The report lists the variant groups, the metrics recorded for the experiment goal, and how each variant compared to the control group. (The significance rating uses the standard statistical significance level of 5%.)

2018–08–01 Page published with no editorial review

2018–08–01 - Service compatible with Unity 5.5 onwards at this date but version compatibility may be subject to change.

New feature in 2018.1

Monetization

Leave feedback

Unity **Analytics** allows you to monitor your in-game revenue from monetization. By implementing receipt verification you'll quickly see legitimate or fraudulent transactions.

Unity Analytics provides the **Analytics.Transaction** method for tracking monetization events through in-app purchases. This method should be called every time a player triggers a monetization event. The **Analytics.Transaction** method requires a price parameter, a currency and an optional Apple iTunes / Google Play receipt string.

```
// Reference the Unity Analytics namespace
using UnityEngine.Analytics;

// Use this call for each and every place that a player triggers a monetizatio
Analytics.Transaction(string productId, decimal price,
string currency, string receipt,
string signature);
```

Analytics.Transaction Input Parameters

Name	*Type*	*Description*
productId	string	The id of the purchased item.
price	decimal	The price of the item.
currency	string	Abbreviation of the currency used for the transaction. For example "USD" (United States Dollars). See here for a standardized list of currency abbreviations.
receipt	string	Receipt data (iOS) or receipt ID (Android) for in-app purchases to verify purchases with Apple iTunes or Google play. Use **null** in the absence of receipts. For more details see Receipt Verification.
signature	string	Android receipt signature. If using native Android use the **INAPP_DATA_SIGNATURE** string containing the signature of the purchase data that was signed with the private key of the developer. The data signature uses the RSASSA-PKCS1-v1_5 scheme. Pass in **null** in the absence of a signature.

The example below is for a $0.99 transaction in USD without receipt validation.

```
Analytics.Transaction("12345abcde", 0.99m, "USD", null, null);
```

Receipt Verification

Leave feedback

Here are the implementation details:

For iOS

receipt parameter

If you leave this as **null**, this transaction will show up in Unverified Revenue
If you are writing a Native **iOS** plugin
For iOS 6 and below, send the raw SKPaymentTransaction's transactionReceipt
For iOS 7 and above, send the base64 encoded value of the App Receipt from the app bundle.
If you are using Unibill plugin
Pass in PurchasableItem's "receipt" property as the receipt.
If you are using Prime31 plugin
Pass in StoreKitTransaction's "base64EncodedTransactionReceipt" property as the receipt.

signature parameter

Pass in **null** since this is not used.

For Android

To validate Android monetization, please enter your Google Public API Key in your **Analytics** dashboard **Project Settings** Form.

The Google Public API Key is needed for implementing receipt verification for in-app purchases (IAP) on Google Play. Your Google Public Key is under Google Play Developer console > All applications > Services & APIs > Your License Key For This Application. This is optional, but if you are developing for Android and have in-app purchases, we highly recommend implementing it.

receipt parameter

If you leave this as null, this transaction will show up as Unverified Revenue
If you are writing a Native Android plugin
Pass in the the purchase data for the order, which is a String in JSON format that is mapped to the INAPP_PURCHASE_DATA key in the response Intent.
If you are using Unibill plugin
Parse the PurchasableItem's "receipt" property as JSON, and pass in the value of the "json" property.
If you are using Prime31 plugin
Pass in GooglePurchase "originalJson" property

signature parameter

If you leave this as **null**, this transaction will show up as Unverified Revenue
If you are writing a Native Android plugin
Pass in the the signature, which is mapped to the INAPP_DATA_SIGNATURE key in the response Intent.
If you are using Unibill plugin
Parse the PurchasableItem's "receipt" property as JSON, and pass in the value of the signature property.
If you are using Prime31 plugin

Pass in GooglePurchase “signature” property.

User Attributes

Leave feedback

Important: reporting user attributes to the Unity **Analytics** service is no longer supported. The `Analytics.SetUserGender()` and `Analytics.SetUserBirthYear()` APIs are deprecated.

2018–06–04 - User Attribute reporting is no longer supported.

Unity Analytics Raw Data Export

Leave feedback

Overview

Unity Analytics Raw Data Export gives you full access to raw event data. This lets you use the data in whatever way you choose, for example to construct custom queries or data visualizations.

There are two ways you can access raw data export:

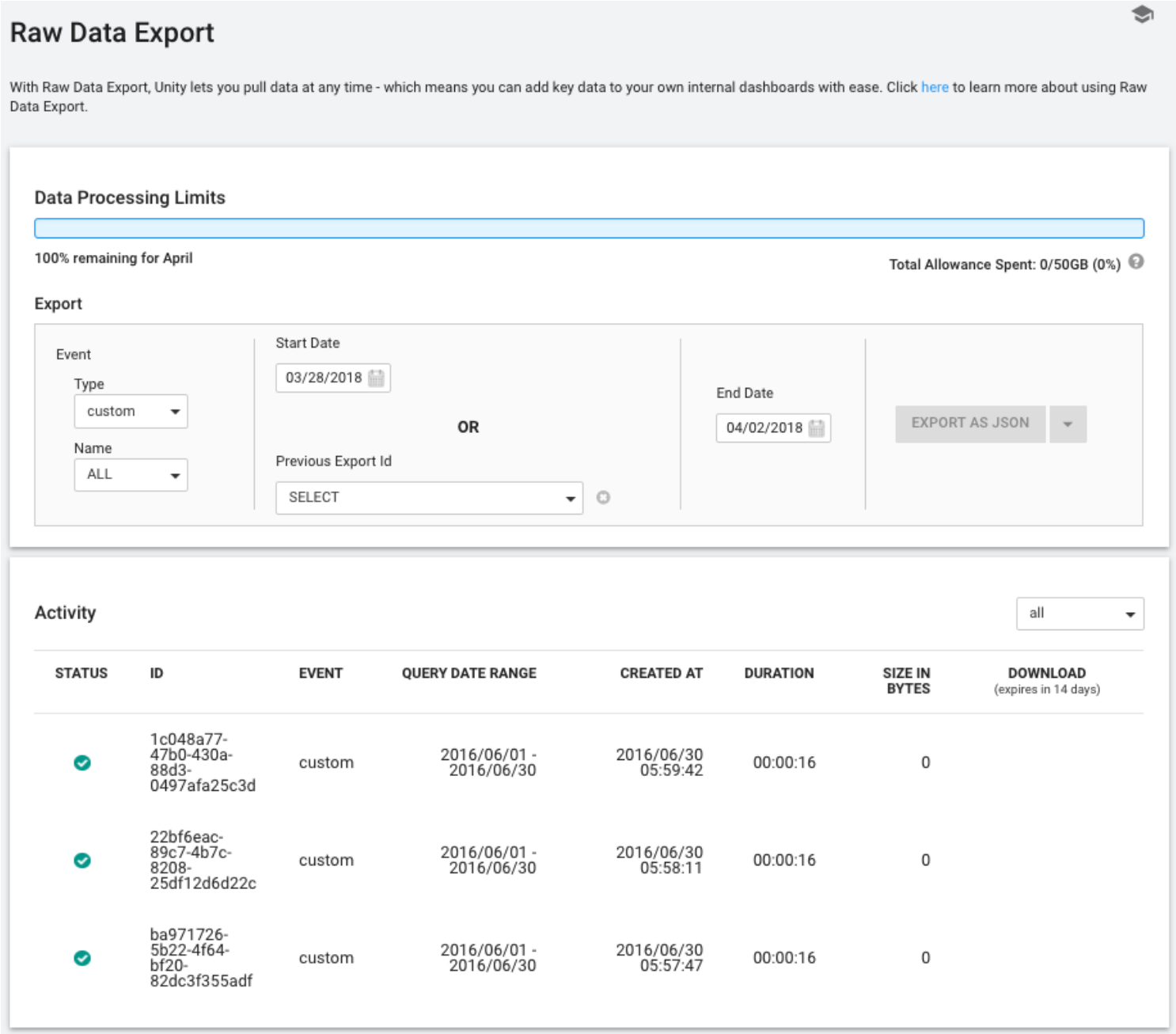
The **Unity Analytics** dashboard user interface
By calling the REST API
Raw Data Export provides data from June 2016 onwards.

Analytics dashboard user interface

The Unity Analytics dashboard provides a way to export and access your raw event data without writing any code. In your Unity Analytics dashboard (analytics.cloud.unity3d.com), select your project and navigate to **Analytics** > **Raw Data Export**.

Then follow these steps in the **Export** section of that screen:

Specify the **Data Set** you want to export (such as appRunning, appStart, custom, deviceInfo, or transaction).
Specify the **Start Date**, or alternatively continue exporting data from a previous job by selecting the **Previous Job Id** from the drop-down box.
Specify the **End Date**.
The file format defaults to JSON. Specify your preferred file format by clicking on and selecting an item from the drop-down menu under **Export as JSON**.



The **Export** section and **Activity** table on the **Raw Data Export** screen

Raw Data Export automatically creates the job and displays it in the **Activity** table on the screen.

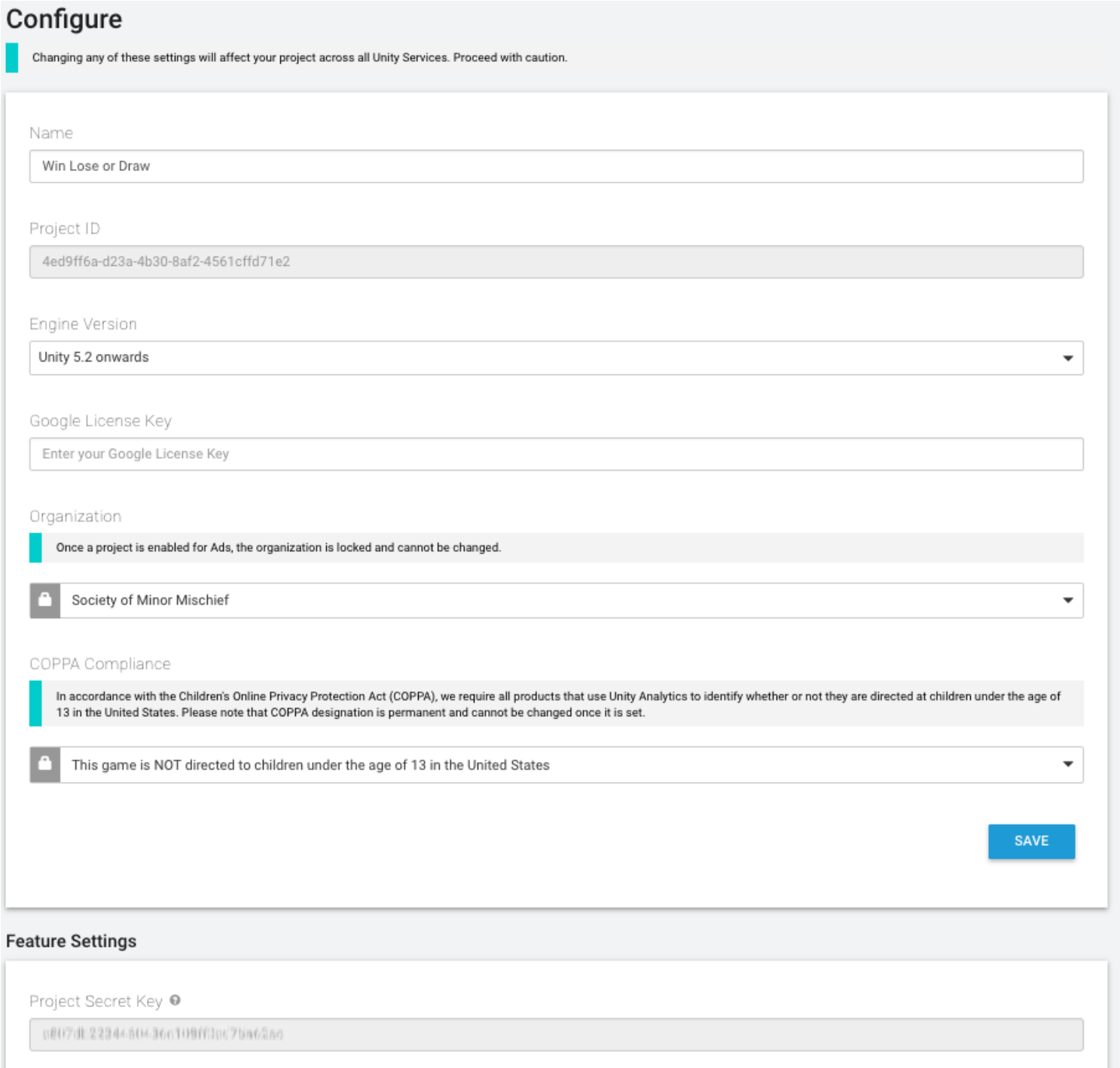
Once the job is done, you can download your data by selecting the files from the **Download** column.

REST (Representational State Transfer) API

Every data point sent to Unity Analytics is stored in Unity's data store. The Raw Data Export APIs allow you to download your raw event data in files as the data is received and stored.

Requirements

Every request requires HTTP Basic authentication using your Unity Project ID (UPID) and API Key.



Limitations

The request period (`startDate` to `endDate`) is limited to 31 days.
Limits to usage may be subject to change.
Unity Analytics retains the raw data files generated for a request for 14 days. (To access the data after 14 days has elapsed, make a new request.)

User workflow

To export raw data, call the Create Raw Data Export API. This request triggers an asynchronous job to process the data. The time this takes depends on how much data is being exported.

To get the current status or result, poll using the Get Raw Data Export API. Once the export is completed, you can get the result in the response of this API. The result contains the list of files and the corresponding download URLs. You can loop through the URLs and download the exported data.

Notes:

All dates and times in requests or responses are in the UTC timezone.
API requests or responses use the JSON format (which is formatted for readability in this document). The data file format is configurable.
Data is exported in .gzip compressed files.
The URL base for the API is https://analytics.cloud.unity3d.com.

Create Raw Data Export

A Raw Data Export is specific to a project and specific to a single dataset (event type). The request period is limited to 31 days.

Use the following HTTP method to create a Raw Data Export:

```
POST api/v2/projects/{UNITY_PROJECT_ID}/rawdataexports
```

Arguments are provided in the payload of the request, and are in JSON format with the Content-Type application/json.

Request argument	Required or optional	Type	Description
startDate	Required unless continueFrom is specified	string	The start date (inclusive) of the export. The date is expressed in YYYY-MM-DD format (ISO–8601).
endDate	Required	String	The end data (exclusive) of the export. The date is expressed in YYYY-MM-DD format (ISO 8601). This is the date at which to close the query. When searching for the current day, use the following day's date.
format	Required	String	The output data format: json (newline-delimited json) or tsv.
dataset	Required	String	One of the following event types: appStart, appRunning, deviceInfo, custom, or transaction.
continueFrom	Optional	String	The Raw Data Export ID to continue exporting data from. This is used for continuing a previous export from the point it finished. See Continuing for more information. Instead of specifying a startDate, a prior Raw Data Export ID could be specified in continueFrom. It is an error to specify both continueFrom and startDate.

Template for a request using cURL on the command line:

```
curl --user {UNITY_PROJECT_ID}:{API_KEY} --request POST --header "Content-Type: application/
https://analytics.cloud.unity3d.com/api/v2/projects/{UNITY_PROJECT_ID}/rawdataexports
```

Example values:

```
UNITY_PROJECT_ID = aa43ae0a-a7a7-4016-ae96-e253bb126aa8
API_KEY = 166291ff148b2878375a8e54aebb1549
REQUEST_JSON = { "startDate": "2016-05-15" , "endDate": "2016-05-16", "format": "tsv", "data
```

An actual request using the example values:

```
curl --user aa43ae0a-a7a7-4016-ae96-e253bb126aa8:166291ff148b2878375a8e54aebb1549 --request
```

The response uses the common Raw Data Export Response Attributes in JSON format.

Raw Data Export Response Attributes

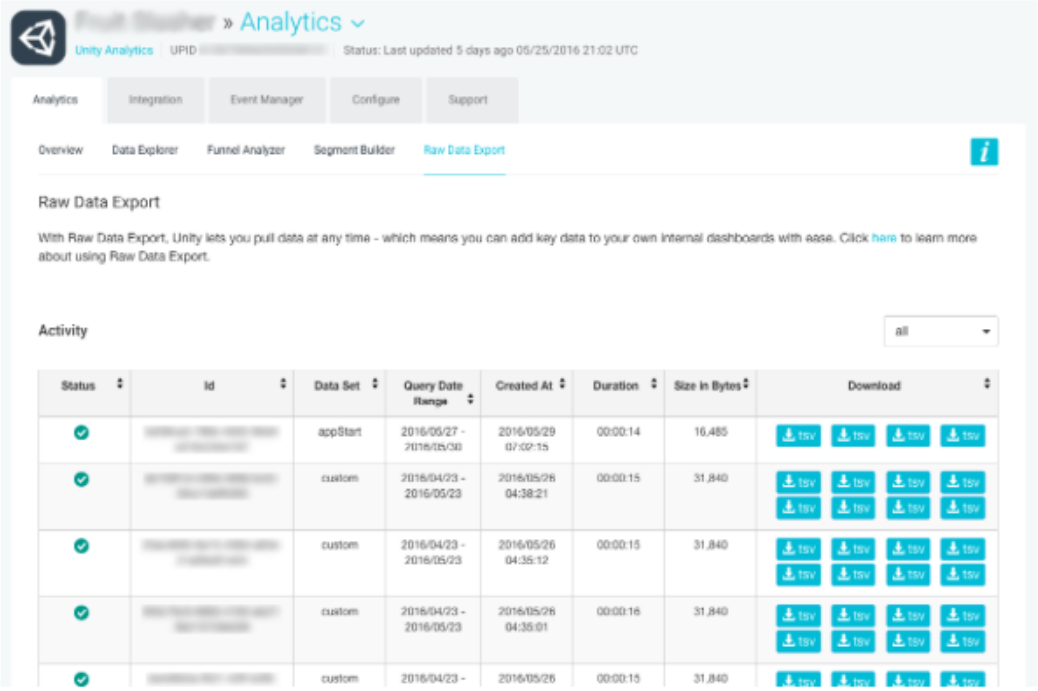
Response attribute	Type	Description
id	String	The Raw Data Export ID.
upid	String	The Unity Project ID.
createdAt	String	The created time in ISO 8601 format.
status	string	The current status of the export. Possible values are: running, completed, or failed.
duration	Long	The time it took to export data (in milliseconds).
request	Json	The request arguments.
result	Json	The result contains attributes that detail the exported data. The result is only available after the export is successfully completed. See below for result attributes.
result.size	Long	The total size of the data exported (in bytes).
result.eventCount	Long	The total number of events exported.
result.intraDay	Boolean	When the request includes the current day it might not contain all the data for the day. This attribute is TRUE if the data is incomplete for the last day.
result.fileList	Json	The list of files containing the data exported. The file list is empty when there is no data.
result.fileList.name	String	The file name.
result.fileList.url	String	The download URL for the file. The files are compressed in gzip format.
result.fileList.size	Long	The size of the file in bytes.
result.fileList.date	String	The file contains events for this specific date. This date is based on the submit time of the event. There may be multiple files for the same date. The date in is in ISO 8601 format.

An example of response:

```
{
   "id":"8228d1e9-31b3-4a5e-aabe-55d9c8afa052",
   "upid":"beff3f49-b9ed-41a4-91ea-677e9b85e71e",
   "createdAt":"2016-05-10T10:10:10.100+0000",
   "status":"running",
   "duration" : 0,
   "request":{
      "startDate":"2016-05-01",
      "endDate":"2016-05-02",
      "format":"json",
      "dataset":"appRunning"
   }
}
```

Continue creating from a prior Raw Data Export

When you are running periodic Raw Data Exports, you must provide the continueFrom argument instead of the startDate to make sure you are continuing from the previous Raw Data Export. Prior Raw Data Export IDs can be fetched via the GET APIs or accessed via the Dashboard.



Prior Raw Data Exports in the Unity Analytics Project Dashboard

Get Raw Data Export

Use the following HTTP method to get a specific Raw Data Export or ongoing export status:

```
GET api/v2/projects/{UNITY_PROJECT_ID}/rawdataexports/{raw_data_export_id}
```

All required arguments are part of the URL path.

An example of a request:

```
curl --user {UNITY_PROJECT_ID}:${API_KEY} https://analytics.cloud.unity3d.com/api/v2/project
```

The response is the Raw Data Export in JSON format. It is the same as the response in Create Raw Data Export.

An example of a response:

```
{
   "id":"6601f70e-6a0b-48ed-909f-26711af82b49",
   "status":"completed",
   "createdAt":"2016-05-21T04:41:54.000+0000",
   "duration":8631714000,
   "request":{
```

```
        "startDate":"2016-02-11T00:00:00.000+0000",
        "endDate":"2016-03-11T00:00:00.000+0000",
        "format":"tsv",
        "dataset":"custom"
    },
    "result":{
        "size":78355,
        "eventCount":17473,
        "fileList":[
            {
                "name":"headers.gz",
                "url":"https://uca-export.s3.amazonaws.com/staging/devTest/custom/appid%3DUNITY_
                "size":105
            },
            {
                "name":"part-4b0cf376-3478-4bc8-845e-f73aff5c0be4.gz",
                "url":"https://uca-export.s3.amazonaws.com/staging/devTest/custom/appid%3DUNITY_
                "size":78250,
                "date":"2016-02-13T00:00:00.000+0000"
            }
        ],
        "intraDay":false
    }
}
```


List all Raw Data Exports

Use this HTTP method to get the list of all Raw Data Exports for a specific project:

```
GET api/v2/projects/{UNITY_PROJECT_ID}/rawdataexports
```

All required arguments are part of the URL path.

An example of a request:

```
curl --user {UNITY_PROJECT_ID}:${API_KEY} https://analytics.cloud.unity3d.com/api/v2/project
```

The response is a list of Raw Data Exports in JSON format. See Raw Data Export Response Attributes for the definition of each export list element.

An example of a response:


```
[
{
```

```
    "id":"6601f70e-6a0b-48ed-909f-26711af82b49",
    "status":"completed",
    "createdAt":"2016-05-21T04:41:54.000+0000",
    "duration":8631714000,
    "request":{
       "startDate":"2016-02-11T00:00:00.000+0000",
       "endDate":"2016-03-11T00:00:00.000+0000",
       "format":"tsv",
       "dataset":"custom"
    },
    "result":{
       "size":78355,
       "eventCount":17473,
       "fileList":[
          {
             "name":"headers.gz",
             "url":"https://uca-export.s3.amazonaws.com/staging/devTest/custom/appid%3DUNITY_
             "size":105
          },
          {
             "name":"part-4b0cf376-3478-4bc8-845e-f73aff5c0be4.gz",
             "url":"https://uca-export.s3.amazonaws.com/staging/devTest/custom/appid%3DUNITY_
             "size":78250,
             "date":"2016-02-13T00:00:00.000+0000"
          }
       ],
       "intraDay":false
    }
 },
 {
    "id":"6601f70e-6a0b-48ed-909f-26711af82b48",
    "status":"completed",
    "createdAt":"2016-05-21T04:41:54.000+0000",
    "duration":8631714000,
    "request":{
       "startDate":"2016-02-11T00:00:00.000+0000",
       "endDate":"2016-03-11T00:00:00.000+0000",
       "format":"tsv",
       "dataset":"custom"
    },
    "result":{
       "size":78355,
       "eventCount":17473,
       "fileList":[
          {
             "name":"headers.gz",
             "url":"https://uca-export.s3.amazonaws.com/staging/devTest/custom/appid%3DUNITY_
             "size":105
          },
          {
             "name":"part-4b0cf376-3478-4bc8-845e-f73aff5c0be4.gz",
             "url":"https://uca-export.s3.amazonaws.com/staging/devTest/custom/appid%3DUNITY_
             "size":78250,
```

```
            "date":"2016-02-13T00:00:00.000+0000"
         }
      ],
      "intraDay":false
   }
}
]
```

TSV format

If you choose to export in TSV format, the headers are provided in a separate file in **headers.gz**. The data files do not include headers.

An example headers file:

```
ts  appid   type    userid  sessionid   platform    sdk_ver debug_device    user_agent  subm
```

Dataset

Each of the six data types (event types) differ. See below for their schema definitions.

Notes:

`ts` is the timestamp at which the event was generated on the device. Note that device-generated timestamps can be skewed due to the device clock and latency in receiving the event.
`submit_time` is the timestamp at which the event was received by Unity Analytics.
If a field in a record has no data, the value of that field is set to the default value for the data type of the field (`0` for numbers, `""` for strings, and `false` for bools). Nested fields, like the IAP `TransactionEvent.receipt` field, in JSON-format exports are an exception to this policy. In the JSON-format export of nested fields, any fields containing no data are not included in the JSON object for that record.

AppStart event

```
{
   "namespace":"com.unity.analytics.commons.schema",
   "name":"AppStartEvent",
   "type":"record",
   "fields":[
       {"name": "ts",   "type": "long", "default": 0},
       {"name": "appid", "type": "string", "default": ""},
       {"name": "type", "type": "string", "default": ""},
       {"name": "userid", "type": "string", "default": ""},
       {"name": "sessionid", "type": "string", "default": ""},
       {"name": "platform", "type": "string", "default": ""},
       {"name": "sdk_ver", "type": "string", "default": ""},
       {"name": "debug_device", "type": "boolean", "default": false},
       {"name": "user_agent", "type": "string", "default": ""},
       {"name": "submit_time", "type": "long", "default": 0}
```

```
    ]
}
```

AppRunning event

```
{
   "namespace":"com.unity.analytics.commons.schema",
   "name":"AppRunningEvent",
   "type":"record",
   "fields":[
       {"name": "ts",   "type": "long", "default": 0},
       {"name": "appid", "type": "string", "default": ""},
       {"name": "type", "type": "string", "default": ""},
       {"name": "userid", "type": "string", "default": ""},
       {"name": "sessionid", "type": "string", "default": ""},
       {"name": "platform", "type": "string", "default": ""},
       {"name": "sdk_ver", "type": "string", "default": ""},
       {"name": "debug_device", "type": "boolean", "default": false},
       {"name": "user_agent", "type": "string", "default": ""},
       {"name": "submit_time", "type": "long", "default": 0},
       {"name": "duration", "type": "int", "default": 0}
   ]
}
```

Custom event

```
{
   "namespace":"com.unity.analytics.commons.schema",
   "name":"CustomEvent",
   "type":"record",
   "fields":[
       {"name": "ts",   "type": "long", "default": 0},
       {"name": "appid", "type": "string", "default": ""},
       {"name": "type", "type": "string", "default": ""},
       {"name": "userid", "type": "string", "default": ""},
       {"name": "sessionid", "type": "string", "default": ""},
       {"name": "platform", "type": "string", "default": ""},
       {"name": "sdk_ver", "type": "string", "default": ""},
       {"name": "debug_device", "type": "boolean", "default": false},
       {"name": "user_agent", "type": "string", "default": ""},
       {"name": "submit_time", "type": "long", "default": 0},
       {"name": "name", "type": "string", "default": ""},
       {
           "name":"custom_params",
           "type":["null",{
               "type":"map",
               "values": ["string","null"],
               "default": ""
```

```
            }],
            "default": null
        }
    ]
}
```

DeviceInfo event

```
{
   "namespace":"com.unity.analytics.commons.schema",
   "name":"DeviceInfoEvent",
   "type":"record",
   "fields":[
       {"name": "ts",   "type": "long", "default": 0},
       {"name": "appid", "type": "string", "default": ""},
       {"name": "type", "type": "string", "default": ""},
       {"name": "userid", "type": "string", "default": ""},
       {"name": "sessionid", "type": "string", "default": ""},
       {"name": "platform", "type": "string", "default": ""},
       {"name": "sdk_ver", "type": "string", "default": ""},
       {"name": "debug_device", "type": "boolean", "default": false},
       {"name": "user_agent", "type": "string", "default": ""},
       {"name": "submit_time", "type": "long", "default": 0},
       {"name": "debug_build", "type": "boolean", "default": false},
       {"name": "rooted_jailbroken", "type": "boolean", "default": false},
       {"name": "processor_type", "type": "string", "default": ""},
       {"name": "system_memory_size", "type": "string", "default": ""},
       {"name": "make", "type": "string", "default": ""},
       {"name": "app_ver", "type": "string", "default": ""},
       {"name": "license_type", "type": "string", "default": ""},
       {"name": "app_install_mode", "type": "string", "default": ""},
       {"name": "model", "type": "string", "default": ""},
       {"name": "engine_ver", "type": "string", "default": ""},
       {"name": "os_ver", "type": "string", "default": ""},
       {"name": "app_name", "type": "string", "default": ""},
       {"name": "timezone", "type": "string", "default": ""},
       {"name": "ads_tracking", "type": "boolean", "default": false}
   ]
}
```

Transaction event

```
{
   "namespace":"com.unity.analytics.commons.schema",
   "name":"TransactionEvent",
   "type":"record",
   "fields":[
       {"name": "ts",   "type": "long", "default": 0},
```

```
        {"name": "appid", "type": "string", "default": ""},
        {"name": "type", "type": "string", "default": ""},
        {"name": "userid", "type": "string", "default": ""},
        {"name": "sessionid", "type": "string", "default": ""},
        {"name": "platform", "type": "string", "default": ""},
        {"name": "sdk_ver", "type": "string", "default": ""},
        {"name": "debug_device", "type": "boolean", "default": false},
        {"name": "user_agent", "type": "string", "default": ""},
        {"name": "submit_time", "type": "long", "default": 0},        {
            "name":"receipt",
            "type":["null",{
                "type":"record",
                "name": "receiptRecord",
                "fields":[
                    {"name": "data", "type": "string", "default": ""},
                    {"name": "signature", "type": "string", "default": ""}
                ]
            }],
            "default": null
        },
        {"name": "currency", "type": "string", "default": "USD"},
        {"name": "amount", "type": "float", "default": 0},
        {"name": "transactionid", "type": "long", "default": 0},
        {"name": "productid", "type": "string", "default": ""}
    ]
}
```

Dictionary

Data field	Definition	Dataset
ts	The timestamp (in milliseconds) at which the event was generated on the device. Note that device-generated timestamps can be skewed due to the device clock and latency in receiving the event	All datasets
appid	The ID that is assigned to each app on the Unity Analytics Dashboard	All datasets
type	The type of event being queried (i.e. Custom, DeviceInfo, Transaction, etc)	All datasets
userid	Unity Analytics generated identifier for a userid	All datasets
sessionid	Unity Analytics generated identifier for the session. If a game is reopened after more than 30 minutes of inactivity, a new session ID is generated	All datasets
remote_ip	The IP address that the session was played from. Note that this field has been removed and is no longer available	Not used
platform	The platform that the session was played on	All datasets
sdk_ver	Version of Unity Analytics SDK that is being used for this event. If the sdk_ver begins with a “u” then it is from the Analytics tool built engine. Otherwise, it is from the Analytics **plugin** for Unity versions older than 5.2	All datasets
debug_device	A boolean that shows whether the event was sent from a **development build**. Set to TRUE for events coming from Unity Editor	All datasets
user_agent	The User-Agent request-header field	All datasets
submit_time	The timestamp (in milliseconds) at which the event was received by Unity Analytics	All datasets

Data field	Definition	Dataset
duration	The duration (number of seconds) that this session has been running for, calculated by the SDK	AppRunning
name	The name of the Custom Event (e.g. “LevelComplete”)	Custom
custom_params	The list of Custom Event parameters and their corresponding values.	Custom
receipt	Includes data returned by platform app stores, such as App Store and Google Play	Transaction
currency	The currency code for the payment (eg. USD, EUR, CAD, etc) in ISO 4217 code	Transaction
amount	The total decimal amount of currency spent	Transaction
transactionid	Unique identifier for this transaction, set by SDK. Each transaction is given a unique ID, not to be confused with the store’s transaction ID	Transaction
productid	The store-specific product identifier of an in-app purchase (eg. com.mygame.100coins)	Transaction
debug_build	A boolean that shows whether the event was sent from a development build. Set to TRUE for events coming from Unity Editor	DeviceInfo
rooted_jailbroken	A boolean that is set to TRUE in case of rooted / jailbroken phone not sent for normal phone	DeviceInfo
processor_type	Device processor type	DeviceInfo
system_memory_size	Device system memory	DeviceInfo
make	Maker of the device (eg. “OSXEditor”)	DeviceInfo
app_ver	Version of the application (eg. “1.0”)	DeviceInfo
deviceid	A unique device identifier. Note that this field has been removed and is no longer available.	Not used
license_type	Type of license (eg. “advanced_pro”)	DeviceInfo
app_install_mode	Tells if app is installed via app store (“store”), adhoc (“adhoc”), developer install (“dev_release”), simulator (“simulator”) or enterprise (“enterprise”)	DeviceInfo
model	Device model (eg. “MacBookPro11,3”)	DeviceInfo
engine_ver	Unity engine version (eg. “5.5.0a3”)	DeviceInfo
os_ver	OS Version (eg. “Mac OS X 10.11.5”)	DeviceInfo
app_name	Bundle identifier or package name (eg. “com.Company.ProductName”)	DeviceInfo
timezone	ISO code (eg. “GMT–7”)	DeviceInfo
ads_tracking	A boolean value that indicates whether the user has limited ad tracking	DeviceInfo
adsid	Advertising Id. On **iOS**, collected when Unity Ads is enabled. On Android, all the time. Note that this field has been removed and is no longer available.	Not used

---

2018–06–28 Page amended with editorial review

2018–08–01 - Removed remote_ip field. It is no longer available.

2018–06–28 - Removed UserInfo event.

2018–06–04 - Removed deviceid and adsid from DeviceInfo event. Deprecated UserInfo event.

2017–06–21 - Inclusion of empty nested fields in JSON exports changed in Unity 2017.1

Data reset

Leave feedback

Unity **Analytics** allows you to permanently clear all data from your **Analytics** dashboard by performing a data reset. This action **cannot be undone**, so read this documentation and the confirmation message in the **Analytics** dashboard carefully before carrying out this action.

This page describes when data reset might be useful, and what impact it has on the data in your Analytics project.

When to perform a data reset

The main scenario that calls for resetting data in an Analytics project is to wipe data from your dashboard immediately prior to launching (or re-releasing) your game. This ensures that the data you collect after the reset is clean and not mixed with previously collected data, which may skew your analyses.

Other common scenarios:

Wiping data collected during development and testing prior to launching your game.

Wiping data from a pre-release or beta version.

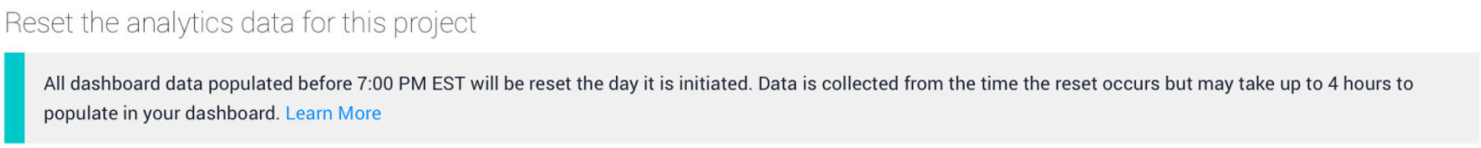
Wiping data prior to launching your game in a store.

Wiping data prior to a product re-launch.

Please note that performing a data reset is optional - you may never need to use this feature.

How this impacts your data

Due to Unity Technologies' data processing cycles, all data resets occur at 00:00 UTC the following day. See the Configure page for the exact time (in your timezone) your data is due to be reset.



Data is collected from the time the reset occurs, but might take up to 4 hours to populate in your dashboard. You cannot specify another date to reset your data on.

Resetting your data affects these Analytics elements as follows:

Dashboard: This is permanently wiped of all visible data, including data in charts.

Custom event data: This is permanently wiped from the dashboard. Note that custom events you have instrumented in your project don't need to be re-validated.

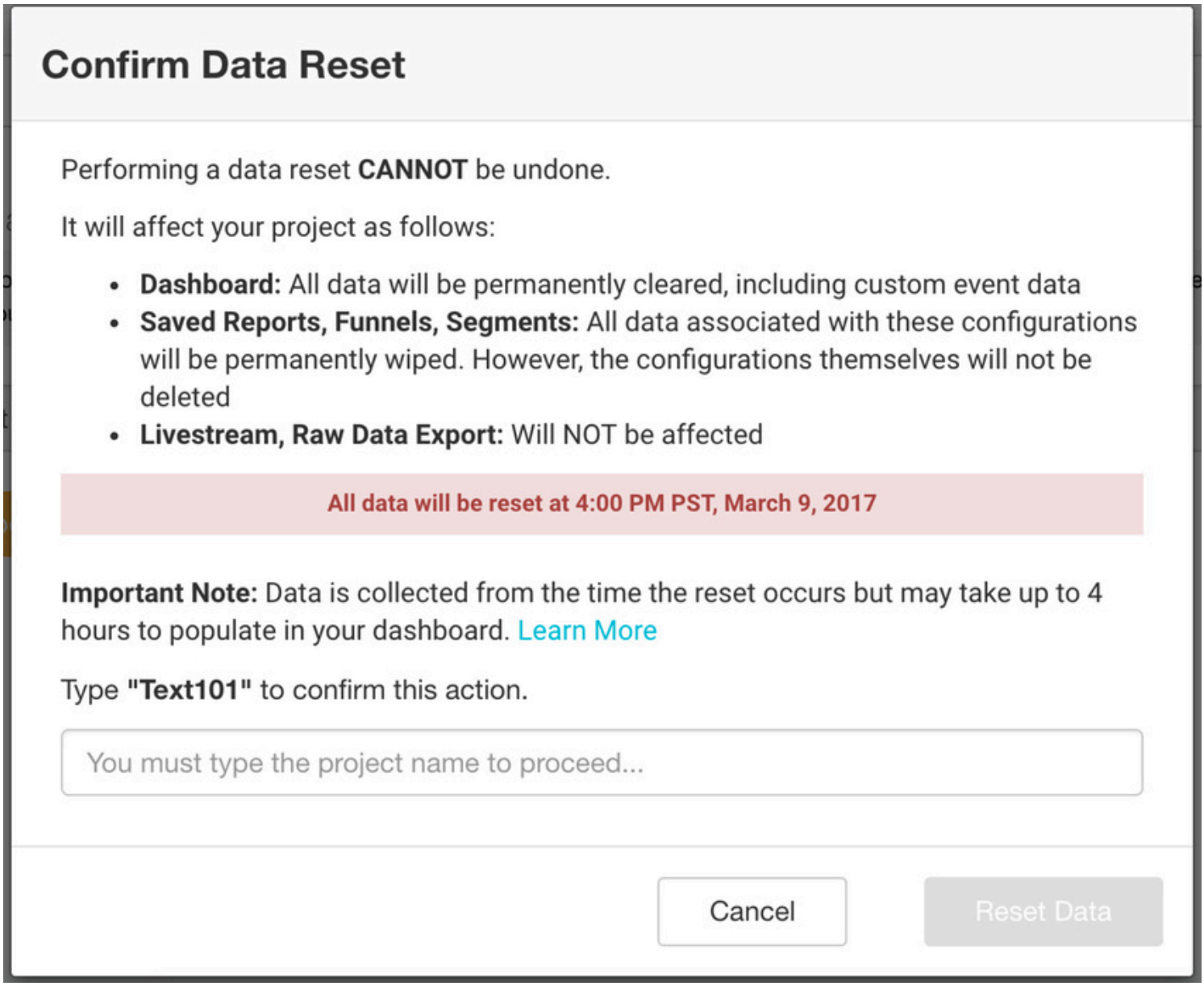
Reports, Funnels, Segments: All data associated with these configurations (that you have created in your project) is permanently wiped from the dashboard. However, the configurations themselves are not deleted. For example, if you create a three-step funnel, populate it with data, and then perform a data reset, the three-step funnel is still present in your dashboard but is no longer populated with the data.

Livestream: This is not affected by performing a data reset.

Raw Data Export: This is not affected by performing a data reset. You can still perform an export to access historical data, and all previous exports you have run are still present in the Activity table on the Raw Data Export page.

How to perform a data reset

To perform a data reset, open the Unity Services Dashboard for the Analytics project that you want to reset, go to **Configure**, and click on **Reset Data**. Make sure to read the confirmation message carefully. Type the name of your project into the text box and click **Reset**. The data reset occurs immediately.



When the data is reset, the following details are provided about the last reset in the **Configure** page:

Date and time the reset occurred.

User who performed the reset.

Note: Only Project Owners and Managers can perform this action. Project Users do not have permission to perform a reset.

Upgrading Unity Analytics

Leave feedback

Unity **Analytics** comes built-in starting with version 5.2 onwards of the Unity Editor. We recommend you upgrade your existing Unity project with **Analytics** already integrated to 5.2 and up. Select the upgrade instructions applicable to you.

Upgrade 4.x–5.0 SDK to 5.2 onwards
Upgrade 5.1 to 5.2 onwards
Re-integrate SDK with 5.1

Note: 5.1 Projects that Integrated Analytics Before September 8, 2015

Unity Analytics SDK will now support 4.x–5.1 versions. The previous 5.1 integration “PlayerSettings” copy-and-paste ProjectID method will still work, but we are discontinuing support for it. Refer to Analytics Forum for more information. For existing 5.1 users, we recommend you update your integration to use the above supported methods.

Upgrade Unity Analytics 4.x–5.1 (SDK) to 5.2 onwards

Leave feedback

The process is divided into three general steps:

Removal of existing Unity **Analytics** SDK **components**

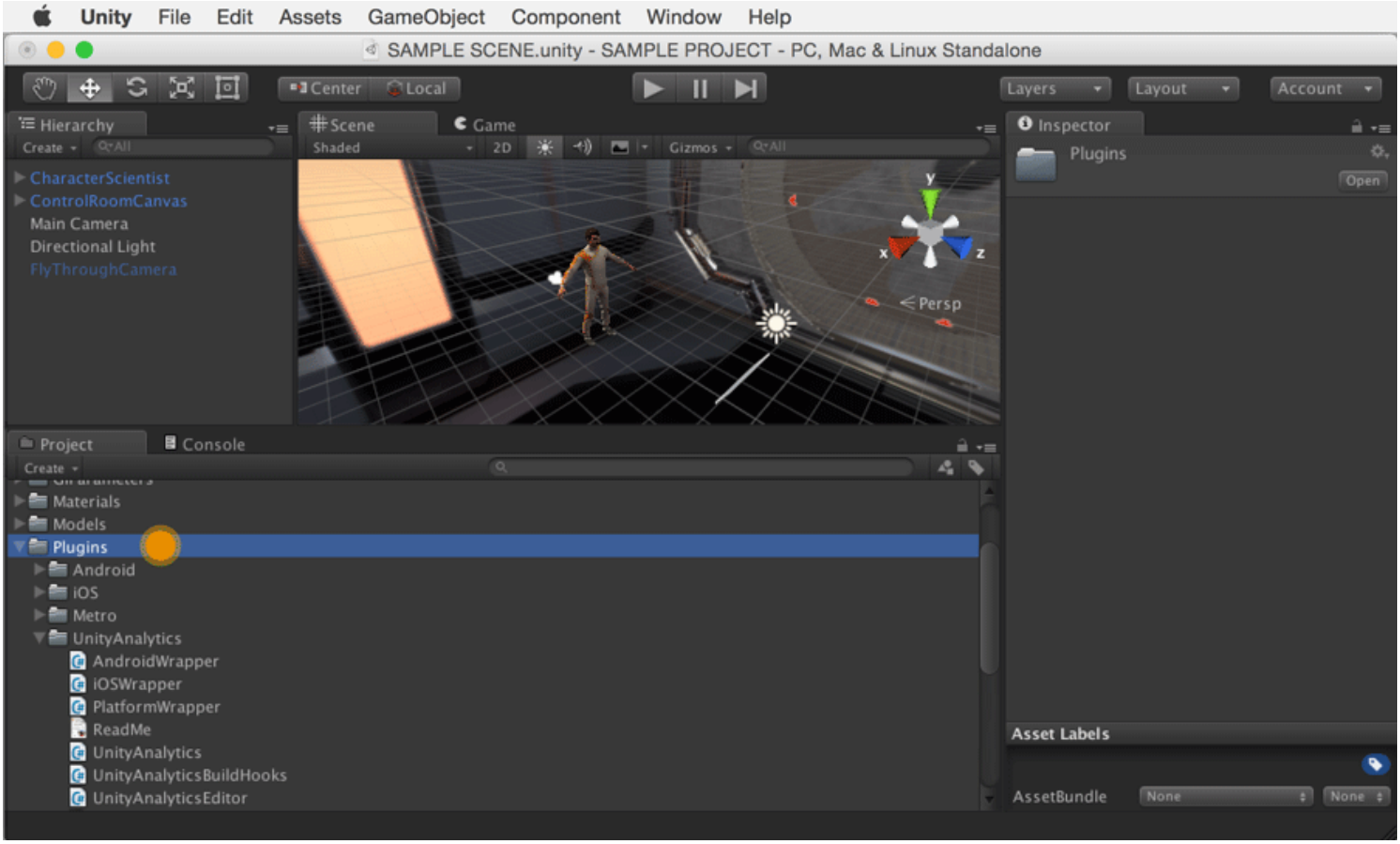
Re-integration of **Unity Analytics**

Updating of Advanced Integration events

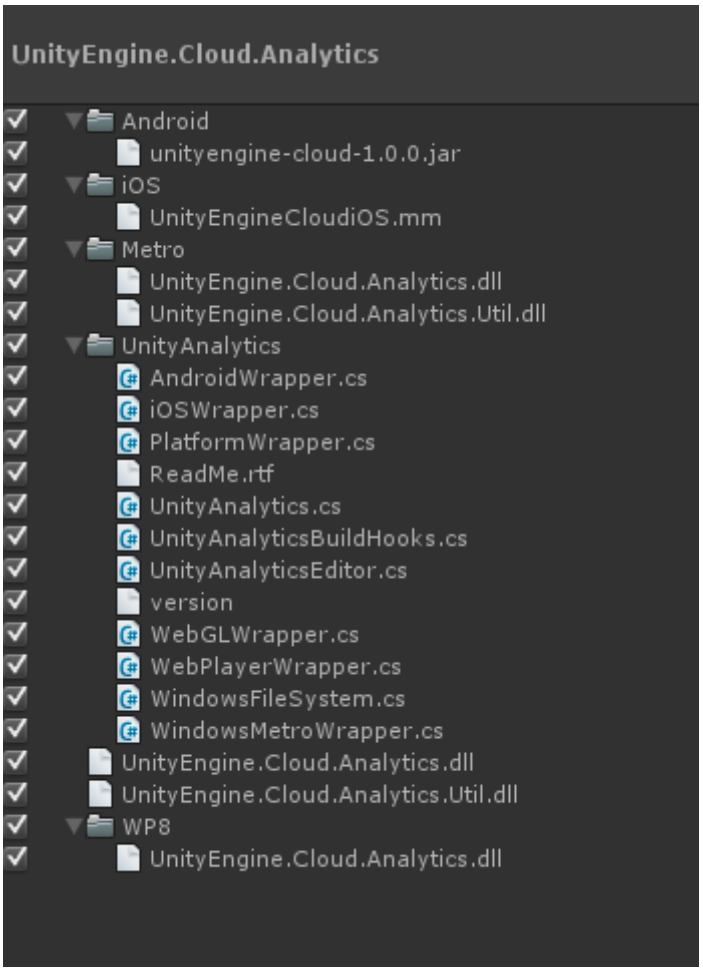
1. Removal of existing Unity Analytics SDK

Version 5.2 onwards of the Unity Editor comes with Unity Analytics built-in. We advise you to remove existing Unity Analytics SDK components to ensure no conflicts and to help you maintain a cleaner project.

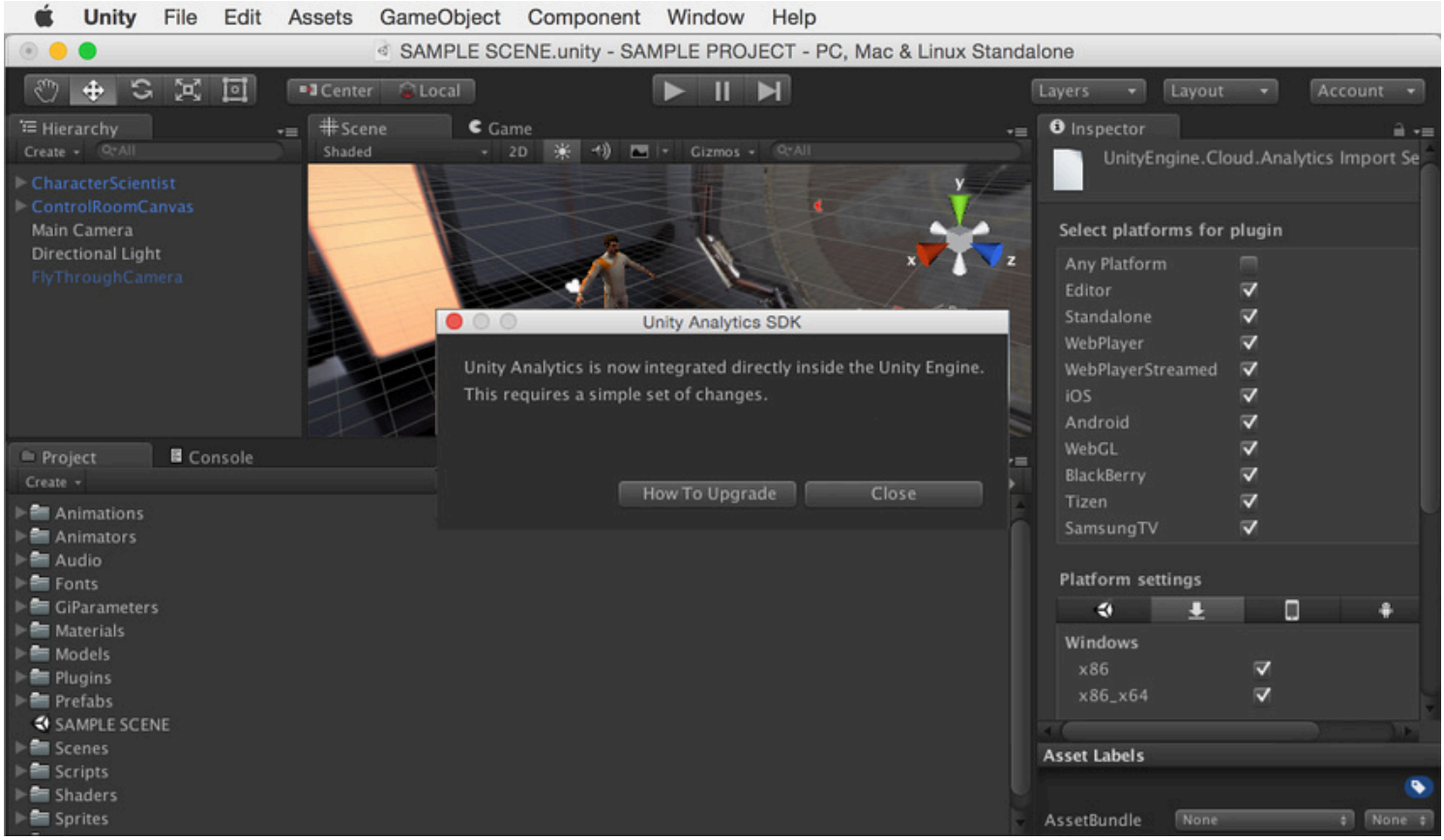
Locate the existing Unity Analytics SDK components in your Project folder. Most likely they will have been saved to Assets > Plugins. **Be sure to only select analytics-specific assets.**



Delete all components.



You can confirm that you successfully removed the Unity Analytics SDK if you close and re-open your project, and then don't see the following message prompt. If you continue to see the prompt confirm all the Unity Analytics SDK components have been removed.



2. Re-integration of Unity Analytics

Follow the instructions in Setting Up Analytics.

You can determine that you successfully completed the upgrade by checking the SDK version in Play to Validate step and confirming that the correct Editor version (ie: u5.2 or u5.3) is displayed in Validator.

UTC Time	Event Type	Detail
07/15/2015 12:37:37	appRunning	**Platform:** W **SDK Versio** u5.2.0b2
07/15/2015 12:37:33	deviceInfo	Device Type: Intel(R) Core(TM) i7-4820K CPU @ 3.70GHz (16325 MB) OS Version: Windows 8.1 (6.3.9600) 64bit Processor: Intel(R) Core(TM) i7-4820K CPU @ 3.70GHz ~~System Memory: 16,325~~ Unity Engine: 5.2.0b2 **Platform:** WindowsEditor **SDK Version:** u5.2.0b2
07/15/2015 12:37:33	appStart	**Platform:** WindowsEditor **SDK Version:** u5.2.0b2

3. Updating Advanced Integration events

If you did any existing Advanced Integration previously, you will also need to update the namespace and the calls to use the 5.2 and up syntax. Follow 5.2 onwards Advanced Integration instructions to update.

Upgrade Unity Analytics 5.1 to 5.2 onwards

Leave feedback

Re-integration of Unity Analytics

Follow the instructions in Setting Up Analytics which consists of these steps:

Confirm your Project IDs match
Enable Analytics
Play to Validate

You can determine that you successfully completed the upgrade by checking the SDK version in Play to Validate step and confirming the correct Editor version (ie: u5.2 or u5.3) is displayed in Validator.

UTC Time	Event Type	Detail
07/15/2015 12:37:37	appRunning	**Platform:** W **SDK Versio** u5.2.0b2
07/15/2015 12:37:33	deviceInfo	Device Type: Intel(R) Core(TM) i7-4820K CPU @ 3.70GHz (16325 MB) OS Version: Windows 8.1 (6.3.9600) 64bit Processor: Intel(R) Core(TM) i7-4820K CPU @ 3.70GHz Unity Engine: 5.2.0b2 **Platform:** WindowsEditor **SDK Version:** u5.2.0b2
07/15/2015 12:37:33	appStart	**Platform:** WindowsEditor **SDK Version:** u5.2.0b2

Updating Advanced Integration events

If you did any existing Advanced Integration previously, you will also need to update the namespace and the calls to use the 5.2 and up syntax. Follow 5.2 onwards Advanced Integration instructions to update.

Unity Analytics Re-integrate SDK to 5.1

Leave feedback

Note: 5.1 Projects that Integrated Analytics Before September 8, 2015 Unity Analytics SDK will now support 4.x–5.1 versions. The previous 5.1 integration "PlayerSettings" copy-and-paste ProjectID method will still work, but we are discontinuing support for it. Refer to Analytics Forum for more information.

The process is divided into three general steps:

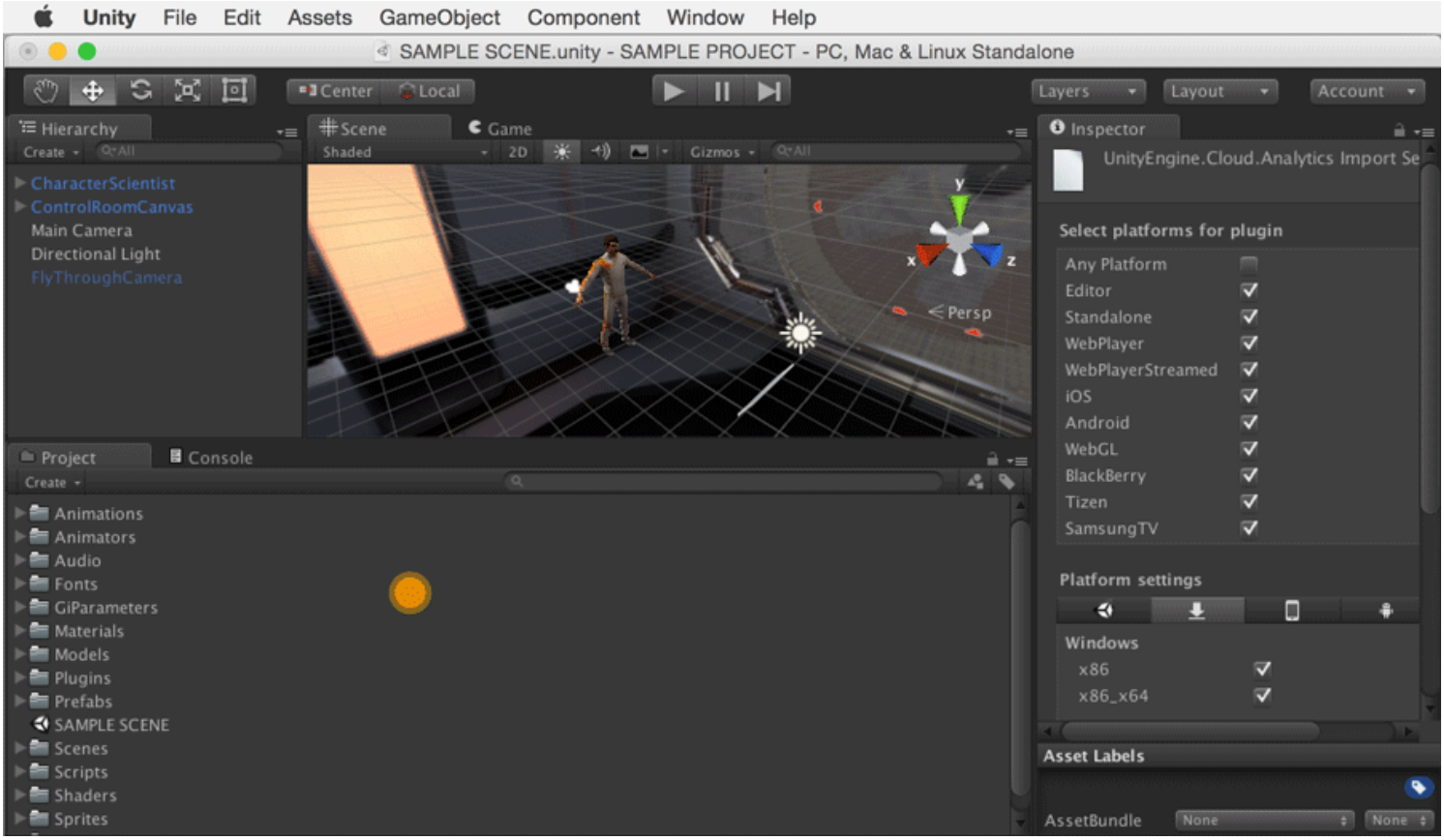
Removal of existing **Unity Analytics** Project ID

Re-integration of Unity Analytics

Updating of Advanced Integration events

1. Remove Existing Unity Analytics Project ID

Open Unity Editor 5.1 and select your existing Analytics project. Locate your Cloud Project ID in **Player Settings**. Choose Edit->Project Settings->Player, which opens the Player Settings. Delete your Cloud Project ID



2. Integrating Unity Analytics SDK

Follow Unity Analytics 4.x–5.1 (SDK) instructions

3. Updating Advanced Integration events

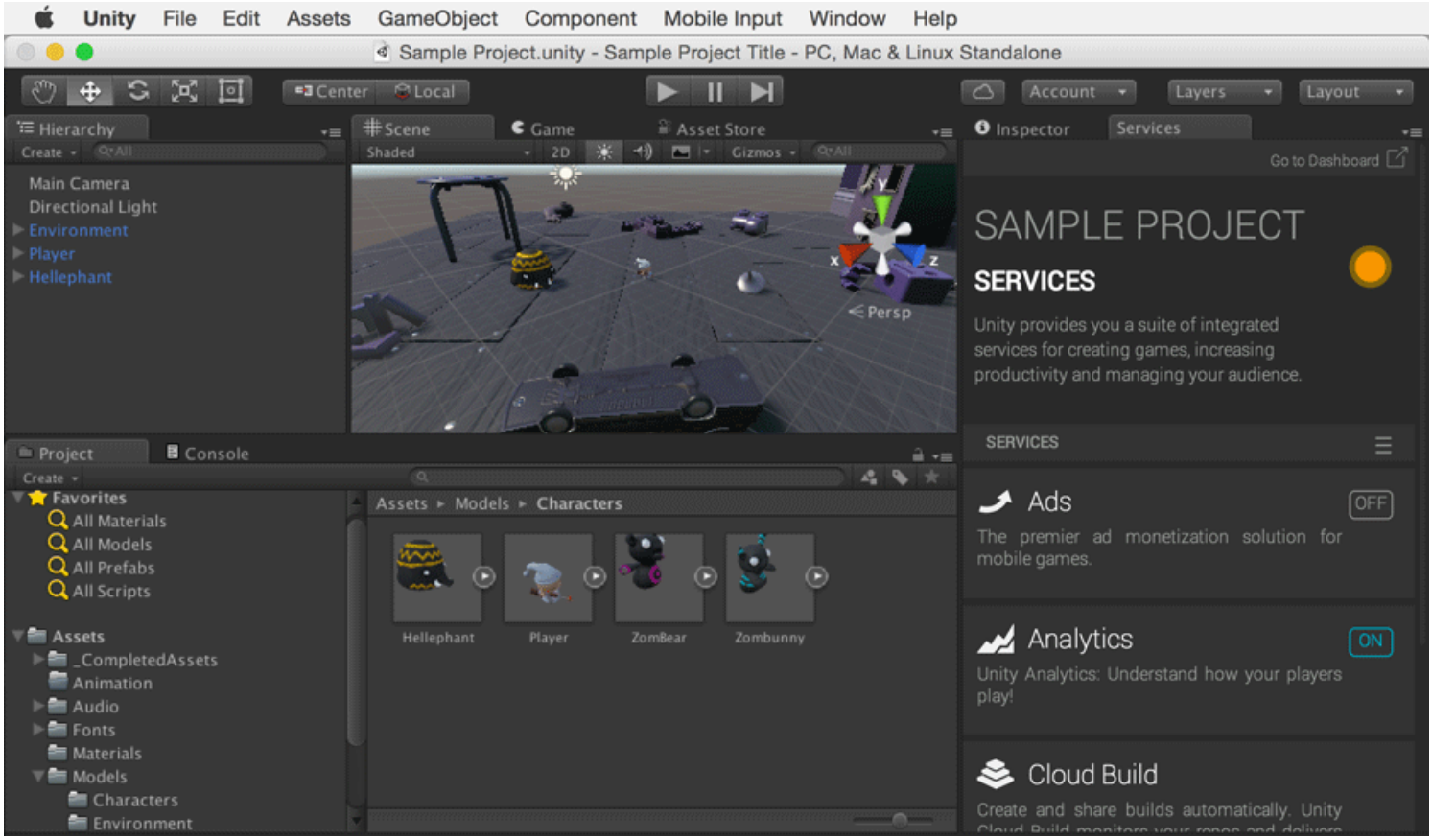
If you did any existing Advanced Integration previously, you will also need to update the namespace and the calls to use the updated 4.x–5.1 SDK syntax.

Follow 4.x–5.1 (SDK) Advanced Integration instructions to update.

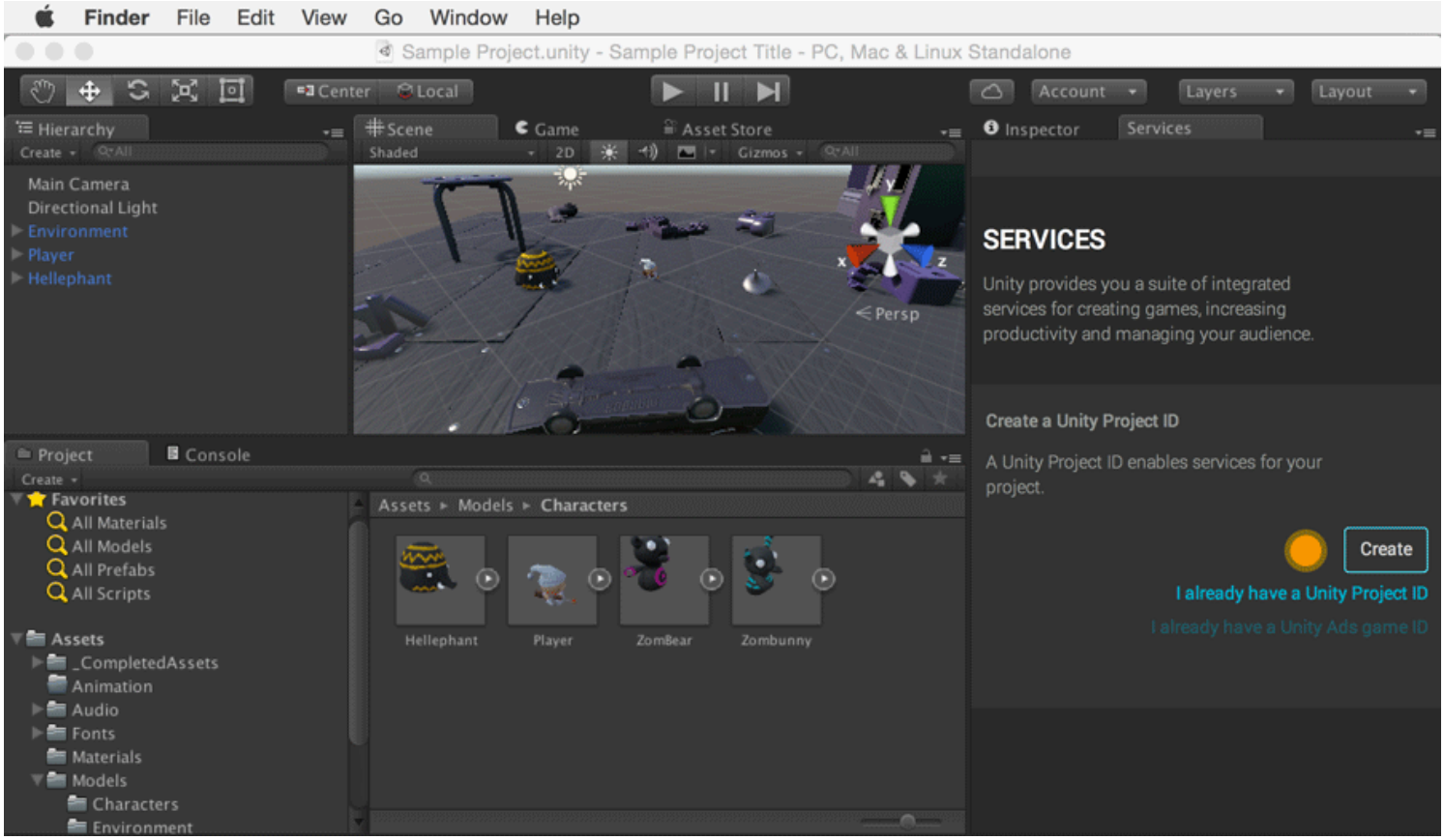
What to Do if Project IDs Don’t Match

Leave feedback

If you cannot find a Project ID in the Services Window Settings, or if you observe a mis-match in Project IDs, this may be due to you upgrading a project using an older version of Unity, or by creating a new project while offline. You can remedy this by linking your **Analytics** project to your project in the Editor.



From **Services Window** > **Settings** > **Unlink project**
Select your Organization and then click “I already have a Unity Project ID”
Find your project in the drop-down, then click “Link” to link your project.


Finder File Edit View Go Window Help
Sample Project.unity - Sample Project Title - PC, Mac & Linux Standalone
Center
Local
Account
Layers
Layout
Hierarchy
Create
Main Camera
Directional Light
Environment
Player
Hellephant
Scene
Game
Asset Store
Shaded
2D
Gizmos
Persp
Inspector
Services
SERVICES
Unity provides you a suite of integrated services for creating games, increasing productivity and managing your audience.
Create a Unity Project ID
A Unity Project ID enables services for your project.
Create
I already have a Unity Project ID
I already have a Unity Ads game ID
Project
Console
Favorites
All Materials
All Models
All Prefabs
All Scripts
Assets
_CompletedAssets
Animation
Audio
Fonts
Materials
Models
Characters
Environment
Assets ▸ Models ▸ Characters
Hellephant
Player
ZomBear
Zombunny

COPPA Compliance

Leave feedback

COPPA stands for the Children's Online Privacy Protection Act (and related regulations). **COPPA** includes certain obligations on operators of applications "directed to children under 13" ("Child Apps"). If you have questions about whether your application is a Child App or other aspects of COPPA, you should refer to materials made available by the U.S. Federal Trade Commission ((portions of which are available here: COPPA FAQ) and consult your own legal advisor.

Unity **Analytics** treats data collected from applications you have designated as a Child App differently from data collected from other applications. If your application is a Child App, you need to designate it as such within the Editor service panel, via the Project creation process in the Unity Ads publisher dashboard or via the Project overview page in the Unity Ads publisher dashboard.

In order to provide analytics for your games, **Unity Analytics** generates an anonymized user ID for each user in your game. We do not use any of these IDs generated from Child Apps to track users across apps built by other developers or to map users between different services, devices, or browsers on the same computer. In addition to these IDs, **Unity Analytics** also collects the following personal information from Child App users: IP address, identifiers for advertisers (IDFA is only collected if Unity Ads is also enabled) and device identifiers (IDFV, Android device ID or IMEI if Android device ID is unavailable).

Unity's use of the personal information identified above is limited to providing app-level analytics for the application's developer and analyzing and reporting anonymized and aggregate-level information about devices, apps and the game industry (for example, the percentage of devices using a certain operating system or the percentage of certain devices by regions). This aggregate-level data contains no personal information. Also, if you've enabled Unity Ads in a Child App, Unity may use information about a user collected by Unity Analytics from that Child App to serve contextual advertising within that Child App.

2017–08–29 Page published with editorial review

Unity Analytics and the EU General Data Protection Regulation (GDPR)

Leave feedback

The General Data Protection Regulation (GDPR) is a European Union law regulating the data privacy of EU citizens.

For more information about EU GDPR, see https://www.eugdpr.org

For information about Unity and GDPR, see https://unity3d.com/legal/gdpr

For Unity's own privacy policy, see https://unity3d.com/legal/privacy-policy

Maintaining compliance with GDPR when you use Unity **Analytics** is a shared responsibility. Unity collects data to help you improve the player experience with ads and gameplay. Some of that data includes personally identifiable information (PII) regulated under GDPR. Unity provides the tools to allow a player to opt out of the PII collection, and to manage the personal data that Unity collects about them, as required by the GDPR. Your responsibilities include adding an opt-out button to your ads, and providing a link to Unity's privacy policy from your own privacy policy.

When your app uses use Unity Ads, Unity displays a notice to the player the first time an ad is shown on their phone, giving them the option to opt in or out of personally identifiable information collection. Subsequent ads also display a button that users can use to manage their data privacy options. For more information about GDPR and the Unity Ads SDK, see Unity Ads Knowledge Base: GDPR Compliance.

If you use both Unity Ads and Analytics, the Unity Ads opt-out mechanism applies to both services.

If you do not use Unity Ads, but do use other Unity services, such as **Unity Analytics**, IAP, Multiplayer, or Performance Reporting, then you must use the **Unity Analytics Data Privacy** plug-in to provide an opt-out choice to your players. The plug-in provides a button you can add to your game that opens a Unity web page where players can manage their privacy settings. Players manage their preferences on a per-game, per-device basis. **Unity Analytics** does not track whether the same player plays more than one game made with Unity, or plays the same game on multiple devices.

The above options cover any data collected by Unity for customizing Ads and player services. However, if you collect personally identifiable information data on your own, you are responsible for protecting and managing that data.

Best practices:

Solicit your own legal advice. Nothing in this document should be construed as legal advice.

Read and understand the Data Processing Addendum (DPA) [PDF]

List Unity as a third-party which collects data in your privacy policy, and include a link to Unity's privacy policy in your privacy policy.

Do not send personally identifiable information to Unity Analytics in Standard or Custom Events.

Do not use the `Analytics.SetUserGender()` or `Analytics.SetUserBirthYear()` to send gender and age information to Unity Analytics. These APIs are deprecated.

Using the Unity Analytics Data Privacy Plug-in

You can obtain the Unity Analytics Data Privacy plug-in from the Unity Asset Store. The plug-in is available for Unity versions:

4.7
5.1+
2017.1+
2018.1+

The plug-in does **not** support the following platforms:

Linux
Windows Phone
Universal Windows Platform (UWP) prior to Unity 5.5
Tizen
Apple TV
Blackberry

The Unity Analytics service now deletes personally identifiable information sent from games running on those platforms automatically. Contact DPO@unity3d.com if you have questions.

If your game displays ads from the Unity Ad network, the Unity Ads SDK already displays a data collection opt-out choice to the player, and configures Unity Analytics based on the player's data privacy choice. You only need to use the Unity Analytics Data Privacy plug-in when you do not use the Unity Ads service.

Configuring Analytics Data Privacy in Unity

If you do not use Unity Ads, then you can integrate the Unity Analytics Data Privacy plug-in, which gives your players control over Unity Analytics data collection.

To integrate the plug-in:

Import the Unity Analytics Data Privacy **Asset** package into your Project.

Provide an opt-out choice to the player using a button or other **UI** control.

The Unity Analytics Data Privacy plug-in contains code that automatically queries the Unity Analytics Service to determine the opt-out status of the current player. The plug-in then configures Unity Analytics based on the player's preferences. The plug-in also includes a Unity UI button **prefab**. This button opens a player's personal data privacy page where they can opt-out of Unity's data collection and view the data that Unity has collected in the past.

Important: If a player has a browser pop-up blocker enabled, their browser can prevent the data privacy page from opening. Some browsers note that a page has been blocked, but others provide no notice at all. Consider adding a message in your user interface that warns players that a pop-up blocker might prevent the page from opening.

See Unity Analytics DataPrivacy API for more information.

Importing the Unity Analytics Data Privacy Plug-in into a project

To import the plug-in into your project:

Go to the Unity Analytics Data Privacy plug-in page on the Unity Asset Store.

Click the **Add to My Assets** button on the page.

In your Unity project, open the **Asset Store** window (menu: **Window** > **General** > **Asset Store**).

Click the **Downloads** icon at the top of the window.

Select **My Purchased**.

Click the **Download** button for the Unity Analytics Data Opt-Out Plug-in.

On the **Import Unity Package** window, click **Import**.

Providing the opt-out choice to the player

The plug-in includes a Unity UI button prefab. When a player clicks this button, the Player Data Privacy page opens in a web browser. You can also provide your own user interface and open the Data Privacy page manually.

Method 1: Using Unity UI

Import the **Asset package** into your project.

If you are building your project with Unity 5.1 or earlier, you must call `DataPrivacy.Initialize()`. Newer versions of Unity call `Initialize()` call automatically.

If you do not already have a Canvas **GameObject** in the **Scene**, add one. Unity automatically adds an EventSystem when you add the **Canvas**.

Drag the **DataPrivacyButton** prefab to the **Canvas** GameObject in the Scene. Find the prefab in the Project window, in the *DataPrivacy/Runtime* folder.

Adjust the position, graphics, and text of the button to suit.

The button is already connected to the Data Privacy API, so that it opens the player's personal data management page when the player clicks it. The page opens in a web browser.

Method 2: Using you own UI

To use your own user interface for the button, you can request the URL of the user's data opt-out page and then open that URL in a browser or web view:

Import the Asset package into your project.

Create your own UI control that informs the player of their ability to opt-out of data collection.

Note: The Data Privacy plug-in includes an icon in the *DataPrivacy/Runtime* folder. Unity encourages you to use this icon on your data privacy button (or similar control) to provide a consistent graphical cue to players encountering data privacy controls in Unity games.

If you are building your project with Unity 5.1 or earlier, you must call `DataPrivacy.Initialize()`. Newer versions of Unity call `Initialize()` automatically.

In response to the player's click or interaction with this control, call the `DataPrivacy.FetchPrivacyUrl()` function. This function takes an `Action<string>` object that invokes when the network request completes. You can optionally pass in a second `Action<string>` function to handle the failure case.

In your handler for the `FetchPrivacyUrl()` request, use `Application.OpenURL()` to open the URL received in a browser.

For example, the following script opens the Player Data Privacy page in response to a click on a GameObject:

```
using System;
using UnityEngine;
using UnityEngine.Analytics;

public class OptOutHandler : MonoBehaviour {

    static void OnFailure(string reason)
    {
        Debug.LogWarning(String.Format("Failed to get data privacy page URL: {0}
    }

    void OnURLReceived(string url)
    {
        Application.OpenURL(url);
    }

    public void OpenDataURL()
    {
        DataPrivacy.FetchPrivacyUrl(OnURLReceived, OnFailure);
    }

    void OnMouseOver(){
        if(Input.GetMouseButtonDown(0)){
            OpenDataURL();
        }
    }
}
```

Using the Data Privacy plug-in with older versions of Unity and Unity Analytics

Before Unity 5.2, Unity provided Analytics support through a Unity Asset Store asset package. You can use the Data Privacy plug-in with these older versions of Unity Analytics, but you must initialize the privacy plug-in before

calling any of its other functions (and before your players have a chance to click the data privacy button). The best place to initialize the plug-in is right after you call `UnityAnalytics.StartSDK()` to initialize the Analytics plug-in:

```
using UnityEngine;
using UnityEngine.Cloud.Analytics;
using UnityEngine.Analytics;

public class UnityAnalyticsIntegration : MonoBehaviour {

    void Start () {
        const string projectId = "SAMPLE-UNITY-PROJECT-ID";
        UnityAnalytics.StartSDK (projectId);
        DataPrivacy.Initialize ();
    }
}
```

Note: Unity 5.2 and later versions call `DataPrivacy.Initialize()` automatically. You do not need to call the function yourself unless you are using the Asset Store package for Analytics support.

See Unity Analytics DataPrivacy API for more information.

2018–05–18 Page published with limited editorial review
Removed Editor menu command for inserting the Data Privacy button. Added the Data Privacy Button prefab.
New feature in Unity 2018.1

Unity Analytics DataPrivacy API

Leave feedback

The `DataPrivacy` class configures the Unity **Analytics** service based on the player's data privacy management choices.

NAMESPACE: `UnityEngine.Analytics`

```
public class DataPrivacy
```

The `DataPrivacy` class automatically fetches the player's data privacy status and configures the Analytics service accordingly.

Use the `FetchPrivacyUrl()` function to fetch the the URL of the player's personal data management page. Open the URL to give the player the option to manage their data privacy settings.

This page details three functions:

Initialize() - Prepares the Data Privacy API for use.
FetchOptOutStatus(Action<bool>) - Fetches the player's current opt-out status and configures the Unity Analytics service.
FetchPrivacyUrl(Action<String>, Action<String>) - Gets the URL of the player's personal data management page.

Initialize()

Prepares the Data Privacy API for use.

Declaration

```
public static void Initialize()
```

Details

This function creates a hidden **GameObject** and adds an instance of the `DataPrivacy` class to it as a **component**.

On Unity 5.1 or earlier, call `Initialize()` early in your application startup (ideally, right after you call `UnityAnalytics.StartSDK (projectId)`. Newer versions of Unity call `Initialize()` automatically.

FetchOptOutStatus(Action<bool>)

Fetches the player's current opt-out status and configures the **Unity Analytics** service.

Declaration

```
public static void FetchOptOutStatus(Action<bool> optOutAction = null)
```

Parameters

[optional] `Action<bool> optOutAction` — The Action object to invoke when Unity has fetched the player's opt-out status. The boolean parameter passed to your Action is `true` if the player has opted out of personal data collection, and `false` if they have not.

Details

The function configures the Analytics service as appropriate for the player's data collection choices. Unity caches the status for use when the player's computer or device is offline.

Unity calls `FetchOptOutStatus()` automatically when your game launches. You should also call the function when the player returns to your app after closing the data privacy management URL, to immediately apply any changes they made while on that page.

You can also use `FetchOptOutStatus()` and pass an `Action<bool>` function parameter to get a player's opt-out choice. The plug-in invokes the `optOutAction` function you provide when the network fetch request is complete. The boolean argument passed to your action is `true` when the player has opted out and `false`, if they have not. If the network request for the player's status fails, Unity calls your `optOutAction` function with the cached value (or `false`, if the status has never been set).

```
void OnOptOutStatus(bool optOut)
{
    if(optOut)
    {
        Debug.Log("Player has opted-out of personal data collection.");
    }
}

// ...
DataPrivacy.FetchOptOutStatus(OnOptOutStatus);
```

FetchPrivacyUrl(Action<String>, Action<String>)

Gets the URL of the player's personal data management page.

Declaration

```
public static void FetchPrivacyUrl(Action<string> success, Action<string> fa
```

Parameters

`Action<String> success` — The Action object to invoke when the URL is successfully retrieved. The string passed to the Action contains the URL.

[optional] `Action<String> failure` — The Action object to invoke when Unity cannot retrieve the URL. The string passed to the Action contains the reason for the failure.

Details

Open the URL passed to your `success` function in a browser or webview to give the player the opportunity to manage their data protection options. Use `Application.OpenURL()` to open the page.

The URL is valid for a short period of time. Always fetch the URL immediately before opening it.

Call `FetchOptOutStatus()` after the player returns to your game to apply any changes immediately. Otherwise, Unity applies any changes to data collection the next time the player launches your game.

2018–05–18 Page published with limited editorial review
Removed Editor menu command for inserting the Data Privacy button. Added the Data Privacy Button prefab.
New feature in Unity 2018.1

Analytics Metrics, Segments, and Terminology

Leave feedback

This glossary defines the metrics and terminology used by the Unity **Analytics** System and documentation.

Metrics

The **Unity Analytics** System collects or computes the following metrics and displays them in reports on the Analytics Dashboard Overview and Data Explorer pages:

Active Player Metrics:

DAU (Daily Active Users)	The number of different players who started a session on a given day. The Unity Analytics system anchors its days on Coordinated Universal Time (UTC), so the DAU figure counts players between 0:00 UTC and 24:00 UTC, no matter which timezone they are located in. A new session is counted when a player launches your game or brings a suspended game to the foreground after 30 minutes of inactivity. DAU includes both new and returning players.
DAU per MAU (DAU/MAU)	The percentage of monthly active users who play on a given day. Also known as Sticky Factor in the analytics and game industries, this metric is often used as one estimate of player engagement. Note that the metric includes both returning and new players. An influx of new players could mask the exit (churn) of existing players. In other words, a steady DAU per MAU metric doesn't mean high player engagement if players are leaving your game just as fast as they join. Always examine retention metrics as well.
MAU (Monthly Active Users)	The number of players who started a session within the last 30 days.
New Users	Users who played your game for the first time. Unity records a persistent, randomized ID value to the PlayerPrefs and uses the id to identify returning players. Note that if the PlayerPrefs file is cleared or deleted, then the player is counted as a new user the next time they launch the game. The PlayerPrefs file is deleted when a player uninstalls and then reinstalls your game (but not when updating).
Number of Users	The cumulative number of unique players over the last 90 days. Users who have not played in more than 90 days are removed from the count.
Percentage of Population	Your player population as a percentage. Typically only useful when combined with a segment. Calculated as the percentage of the Number of Users metric who are members of a specified segment.

Session Metrics:

Sessions per User	The average number of sessions per person playing on a given day. Also known as Average Number of Sessions per DAU.
Total Daily Playing Time	The cumulative playing time of all people playing on a given day.
Total Daily Playing Time per Active User	The average playing time of people playing on a given day.
Total Sessions Today	The total number of sessions by all people playing on a given day. Also known as Total Sessions.

Retention Metrics:

Unity Analytics calculates retention using the method known as *N-Day Retention*. Mathematically, N-Day Retention is calculated as the number of people playing exactly N days after their first session, divided by the number of new users exactly N days ago.

The standard retention metrics calculated by Unity Analytics include:

Day 1 Retention	The percentage of players who returned to your game one day after playing the first time.
Day 7 Retention	The percentage of players who returned to your game seven days after playing the first time.
Day 30 Retention	The percentage of players who returned to your game thirty days after playing the first time.

Monetization Metrics:

ARPDAU (Average Revenue Per Daily Active User)	The average revenue per user who played on a given day. Revenue includes money earned from Unity Ads, verified Unity IAP transactions, and verified IAP transactions reported using the Analytics.Transaction() function.
ARPPU (Average revenue Per Paying User)	Average verified IAP revenue per user who completed a verified IAP transaction.
Number of Unverified Transactions	The total number of IAP transactions, whether or not they have been verified. IAP transactions include Unity IAP purchases and purchases reported using the Analytics.Transaction() function.
Number of Verified Transactions	IAP transactions that have been verified through the appropriate app store. IAP verification is currently supported by the Apple App Store and the Google Play Store.
Total IAP Revenue	The total IAP revenue, including revenue from both verified and unverified transactions. IAP transactions include Unity IAP purchases and purchases reported using the Analytics.Transaction() function.
Total Verified Revenue	Revenue from Unity Ads and verified IAP transactions. IAP verification is currently supported by the Apple App Store and the Google Play Store.
Unverified IAP Revenue	IAP revenue from sources that do not support verification and from transactions that failed verification. Transactions can fail verification because they are fraudulent or because of missing or malformed information. For example, if you haven't configured your Google API key on the Analytics Dashboard, attempts to verify Google Play Store transactions will fail. See Receipt Verification.
Verified IAP Revenue	Revenue from verified IAP transactions. IAP verification is currently supported by the Apple App Store and the Google Play Store.
Verified Paying Users	Players who made verified IAP purchases. IAP verification is currently supported by the Apple App Store and the Google Play Store.

Ad Metrics:

Ad ARPU (Average Revenue Per User)	Average Unity Ads revenue per player.
Ad Revenue	Total Unity Ads revenue.
Ad Starts	The number of video ads that started playing.
Ads per DAU	The number of ads started per active player on a given day.
eCPM (estimated Cost Per Mille)	The estimated revenue for 1000 ad impressions for your app. See What is ECPM.

For more information about Ads, see the Unity Ads Knowledge Base.

Segments

Segments are subsets of your player base, split apart by key differentiators, such as country, platform, experience level, or spending patterns. Segments are used with the **Data Explorer**, **Funnel Analyzer**, and **Remote Settings** features.

The Analytics Service provides a set of standard segments and provides the ability to create your own segments. You can view the rules defining the standard segments and create your own custom segments on the Analytics Dashboard Segment Builder page.

Life Cycle Segments:

The life cycle segments group players by the number of days since they first played your game. The service defines the following life cycle groups:

1–3 Days
4–7 Days
8–14 Days
15–30 days
31+ Days

Note: The life cycle segments only include players in the first 90 days of playing your game. In other words, the **31+ Days** segment includes players who first played more than 30 days ago and less than 90 days.

Geographical Segments:

The geographical segments group players by country. Players are geolocated using techniques such as IP analysis. The standard geographical segments include the following countries:

Australia
Canada
China
France
Germany
Japan
Russia
South Korea
Taiwan
United Kingdom
United States
Rest of World

The *Rest of World* segment includes all players from countries not listed individually. Note that you can define your own custom segments to group players by other, individual countries. (However, those players are also included in the Rest of World standard segment.)

Monetization Segments:

The monetization segments group players by how much money they spend on in-app purchases in your game:

Whales	Players who have spent at least $20 in their lifetime.
Dolphins	Players who have spent between $5 and $19.99.
Minnows	Players who have spent less than $5 in their lifetime.
Never Monetized	Players who have never spent real currency.
All Spenders	Players who had made any verified or unverified in-app purchases in their lifetime.

The *Whales, Dolphins, Minnows* metaphor is a common analytics analogy based on the observation that a small number of users have a disproportionately large impact on the monetary “ecosystem” of a typical F2P game.

Platform Segments

The platform segments group players by the platform on which they are playing. The Analytics system provides two standard platform segments:

Android Users

iOS Users

Custom Segments

Create custom segments to gain more fine-grained segmentation than that provided by the standard segments, as well as to create segments that combine rules from different segment categories.

Unlike standard segments, once a player becomes a member of a custom segment, the player is always a member.

Standard and Custom Events	Segment by whether a specific custom event has been dispatched while a user is playing. You can also specify restrictions on parameter values.
Sessions	Segment by when and how long a user played.
IAP revenue	Segment by IAP activity.
Demographics	Segment by reported demographics.
Geography	Segment by country.
Application version	Segment by application version or bundleid.

New segments apply to new data only. Existing data is not re-processed using the new segmentation rules.

See Segment Builder for more information about defining your own segments.

Terminology

The following terms are used by the Unity Analytics Service:

Analytics Events	Events dispatched to the Analytics Service by instances of your applications. Analytics events contain the data that is processed and aggregated to provide insights into player behavior.
Cohort	A group of players with at least one similar characteristic. You can define and analyze different cohorts of your userbase with segments.
Core Events	Core events are the basic events dispatched by the Unity Analytics code in your game. These events, and the analytics based on them, become available simply by turning on Unity Analytics for a project. Core events include: app running, app start, and device info.
Standard Events	Standard events are application-specific events that you dispatch in response to important player actions or milestones. Standard events have standardized names and defined parameter lists.
Custom Events	Custom events are freeform events that you can dispatch when an appropriate standard event is not available. Custom events can have any name and up to ten parameters. Use standard events in preference to custom events where possible.
Funnel	A funnel is a linear sequence of standard or custom events that you expect a player to complete in order. Analysis of funnels can reveal those steps in the sequence where players encounter friction or problems. For example, a funnel based on level completion events would reveal whether larger than expected numbers of players stopped playing at specific levels. See Funnels.
Heatmaps	Heatmaps are a spatial visualization of analytics data. The heatmap visualization displays markers for special analytics events overlaid on your **scene** in the Unity Editor.Heatmaps requires access to Raw Data Export, a pro-subscription feature. See Heatmaps.
Remote Settings	Remote settings are game variables that you can set remotely on your Analytics Dashboard. See Remote Settings.
Aggregated	As events are received by the Analytics Service, the data is processed and aggregated. The

Data	aggregated data is the basis of most of the Dashboard reports and visualizations.
Raw Data	The individual Analytics events sent by your application. Developers with Unity Pro subscriptions can download this data using Raw Data Export for analysis in external tools or other purposes.
Segment	Segments are subsets of your player base, split apart by key differentiators. Viewing metrics and events by segment can reveal differences in-game behavior between different groups.
Session	A single play or usage period. A new session is counted when a player launches your game or brings a suspended game to the foreground after 30 minutes of inactivity.

The following general terms are helpful to understand:

Active Users	Players who recently played your game. Unity Analytics defines an active player as someone who has played within the last 90 calendar days.
Churn	The rate at which users are leaving your game during a specified period. Your user churn is important in estimating the lifetime value of your users. Mathematically, churn is the complement of retention (in other words: Churn + Retention = 100%).
COPPA (Children's Online Privacy Protection Act)	COPPA is a US law that applies to apps that collect personal information and are targeted to children under the age of 14. See COPPA FAQ and COPPA Compliance
Conversion Rate	The percentage of users who complete an action or sequence of actions. For example, the conversion rate from one step of a funnel to the next is the percentage of players who make it through both steps, while the conversion rate of the funnel as a whole is the percentage of players starting the funnel who complete the entire sequence. The term conversion is also used in other contexts, such as starting and then completing a purchase, or watching an ad and then clicking on a link in the ad.
CTR (Click Through Rate)	The percentage of players who click a link in an ad displayed in your game.
Engagement	Engagement is a broad measure of how players enjoy, or are otherwise invested, in your game. Impossible to measure directly, the following metrics are frequently used to estimate engagement: Retention, DAU, MAU, DAU/MAU, number of sessions, and session length.
F2P (Free to Play)	A business model that offers users free access to a fully functional game and a significant portion of app content. Monetization strategies for these titles generally include microtransactions that allow users to access premium features and virtual goods.
Fill Rate	The rate at which ads are available when you request one.
GDPR (General Data Protection Regulation)	A European Union law regulating data privacy. See Unity Analytics and the EU General Data Protection Regulation for more information.
IAP (In-app Purchase)	Revenue from "micro-transactions" within a game.
Impressions	The number of times ads are seen in your game. An impression is counted even if the ad is not completed.
LTV (Lifetime Value)	The estimated value of an average player over their lifetime with your application or game.
Sticky	An estimate of how compelling a game is to its players. A high "sticky factor" means that players

Factor	stick with an app over time. Unity Analytics calculates Sticky Factor as the percentage of monthly active users who played on a particular day (DAU/MAU), which is a commonly used definition. Note that this definition can hide the exit of existing users with an influx of new users, so it should not be relied on as an indicator of the “health” of your game in isolation from your retention metrics.

2017–08–29 Page published with editorial review

2018–06–04 - Removed Demographics Segments, which are no longer supported.

New feature in Unity 2017.1

Unity Cloud Build

Leave feedback

Unity **Cloud Build** is part of Unity teams and provides continuous integration services for Unity projects. It helps you save time by automating the process of creating builds on Unity's servers. This automatic process frees your team and your team's computers from the time-consuming process of making builds. It helps you catch problems sooner, share builds with your collaborators, and iterate versions of your development more rapidly.

Unity Cloud Build can build your project for multiple platforms, and supports many of the popular version control systems. See Supported platforms to learn more.

Find additional information about **Unity Cloud Build** on the following pages:

Unity Cloud Build: Frequently asked questions
Unity Cloud Build: Support Knowledge Base
Tutorial: Getting Started with Unity Cloud Build

Automated Build Generation

Leave feedback

Automated Build Generation is a feature of **Cloud Build** for Unity Teams Advanced. It provides continuous integration by automating the publishing of changes and building your Project whenever a team member makes changes to the project. When a team member commits a change, **Cloud Build** uses the latest code from the shared repository to build your Project.

The goal of Automated Build Generation is to compile your project every time there is a change. This gives you the most accurate idea of when and where errors occur and ensures you always have a build of your game based on the last good commit.

How Automated Build Generation works

Unity Cloud Build connects to either Unity Collaborate or your source control system, and monitors that system for changes to your Project. When it detects changes to your Project, it downloads your project and builds it for all of your target platforms. When the builds finish, **Unity Cloud Build** emails you the results, along with links to download and install the builds. If there are errors, **Unity Cloud Build** informs you straight away, allowing you to quickly fix them, commit the changes, and trigger new builds.

2018–04–10 Page amended with editorial review

Supported platforms

Leave feedback

Unity **Cloud Build** allows you to build your Unity project for multiple platforms and operating systems. The system currently supports:

iOS

Android

Windows desktop

Mac OS desktop

Linux desktop

WebGL

Unity Web Player (Unity 5.3 and below, legacy versions of Unity only)

Supported versions of Unity

Leave feedback

Unity **Cloud Build** can build projects created with most versions of the Unity Editor. **Unity Cloud Build** is usually available to new versions of the Unity Editor within 48 hours of the new version release.

Visit the Unity Cloud Build website's guide to Supported Unity versions to see a full and up-to-date listing of all versions of Unity supported.

Version control systems

Leave feedback

In order for **Unity Cloud Build** to make builds for you, you have to first host your project in Unity Collaborate or a third-party source control system (also known as a “version control system”). Unity **Cloud Build** currently supports:

Unity Collaborate

Git

Apache Subversion (SVN)

Mercurial

Perforce

Using the Unity Developer Dashboard to configure Unity Cloud Build for Git

Leave feedback

Unity Cloud Build supports projects stored in Git repositories. Your repository can be hosted on GitHub, GitLab, Bitbucket, or private servers.

Note: This feature requires a subscription to Unity Teams Advanced. For more information, see the Unity Teams page.

This topic covers:

Configuring GIT in the Dashboard

Using private repositories

Setting up a target build platform

Using Git submodules

Configuring Git on the Dashboard

To configure Cloud Build to build your Project from a GitHub repository:

Sign in to the Unity Developer Dashboard.
On the **Projects** page, select your Project.
In the dashboard **Overview** window, on the **Cloud Build** tile, click **OPEN CLOUD BUILD DASHBOARD**.
In the **Build History** window, click **Set up Cloud Build**.
You can configure access to your repository using the following authorization protocols:

OAuth (default)
SSH
To configure Cloud Build to use OAuth to access your repository:

In the **Source Control** window, click the **GitHub**, **Bitbucket**, or **GitLab** tile.
Authorize access to your account as appropriate for the selected source code control service.
In the **Select a Repository** window, choose a repository from which to build the Project and then click the **NEXT: TARGET SETUP** button.
Note: If you are currently signed into the source code management service, Cloud Build uses your current credentials to retrieve the repository list. To configure a Project that retrieves a repository from a different account, sign out of the source code management service first, then configure the Project.

For the next step, see the Setting up a target build platform section.

To configure Cloud Build to use SSH to access your repository:

In the **Source Control** window, click the **Manual** tab.

In the **SCM URL** field, enter the URL of your Git server in either of the following formats:

Non-SSL: host:port
SSL: ssl:host:port
To connect to your repository, you must specify the URL to your Git server. For information on the format of the URL, see the **URL syntax** section below.

From the **SCM Type** drop-down menu, select **GIT**.

Click the **NEXT: ACCESS** button.

When Unity Cloud Build connects to the hosting site, it automatically detects whether your repository is public or private. If your repository is public, Cloud Build automatically connects to it and you can skip to Setting up a target build platform. If your repository is private, see the Using private repositiessection.

When Unity Cloud Build connects to the hosting site, it automatically detects whether your repository is public or private. If your repository is public, Cloud Build automatically connects to it and you can skip to Setting up a target build platform.

URL syntax

To connect to your repository, you must specify the URL to your Git server. You can specify the URL to use the following protocols:

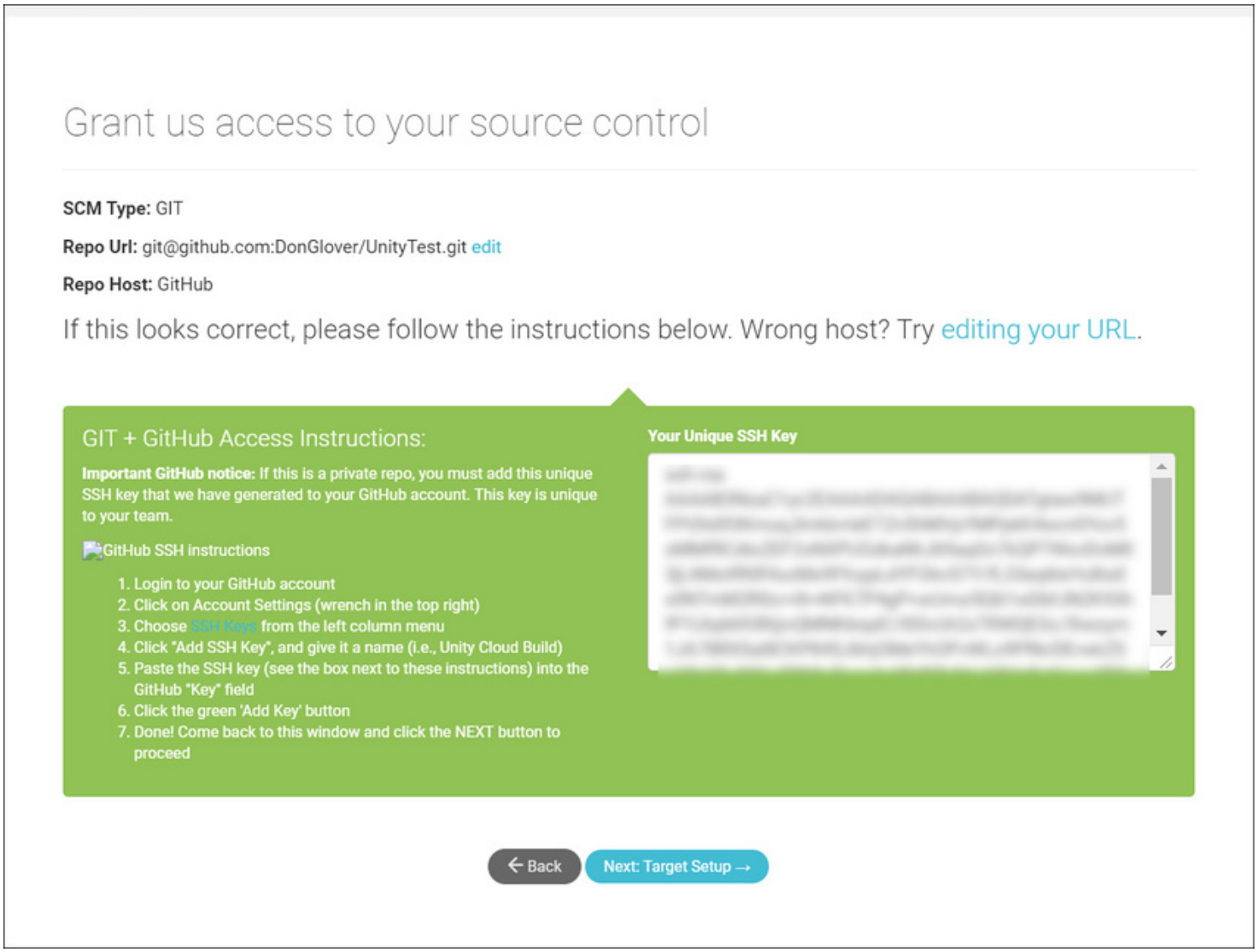
HTTPS
GIT
SSH
The following are examples of URLs for GitHub, bitbucket, and GitLab:

https://github.com/youraccount/yourrepo
git://github.com/youraccount/yourrepo.git
git@bitbucket.org:youraccount/yourrepo.git
git@gitlab.com:youracccount/yourrepo.git
Note: If you are hosting Git on a private server, you must use SSH to connect to your repository.

Use the format that is most convenient for you.; Unity Cloud Build automatically re-writes the URL into the format it needs.

Using private repositories

If your repository is private, Cloud Build must use SSH to connect to it. When Cloud Build detects that the repository is private, it displays the **Grant us access to your source control** screen.



Setting your SSH key on GitHub

To add your SSH key to your project in GitHub

Sign in to GitHub.

In the upper-right corner of any GitHub page, click your profile photo, then click Your profile.

On your profile page, click **Repositories**, then click the name of your repository.

In your repository, click the **Settings** tab.

In the sidebar, click **Deploy Keys**, then click the **Add deploy key** button.

In the **Title** textbox, type name to identify this key.

In the **Key** field, paste in your public key from the **Grant us access to your source control** window, and then click the **Add key** button.

For more information on deploy keys, see Managing Deploy Keys in the GitHub documentation.

You can also add your SSH key to your GitHub account so that all repositories in your account are accessible to Cloud Build. For more information, see Adding a new SSH key to your GitHub account in the GitHub documentation.

Setting your SSH key on Bitbucket

To add your SSH key to your Bitbucket account:

Log in to Bitbucket.

Click your **avatar** in the lower left of the page.

Click **Bitbucket Settings**.

On the **Settings page**, in the **SECURITY** section, click **SSH Keys**.

In **SSH Keys**, click the **Add key** button.

In the **Label** field, enter a recognizable name for the key (such as **Unity Cloud Build**.)

Paste the Unity Cloud Build SSH key from the **Grant us access to your source control** window into the **Key** field.

Click **Add Key**.

On the **Grant us access to your source control** window, click **Next: Target Setup**.

Setting up a target build platform

In the dashboard, on the **NEW BUILD TARGET: BASIC INFO** window:

In the **Target Label** field, enter a name for the build.

If the root folder of your repository doesn't contain your **Assets** and **Project settings**, in the **Project Subfolder** field, enter the URL of the folder that contains your **Assets** and **Project settings**.

From the **Unity Version** drop-down menu, select the version of Unity with which to build the Project.

If you do not want the Project to automatically build whenever your repository is updated, click the **Auto-build** toggle to disable this feature.

If you are building for the **iOS** or Android platforms, your next step is to enter credentials for the build. Click **Next: Credentials**. For all other platforms, click **Next: Build** to complete the configuration and start the initial build.

Android credentials

In the IOS **SIGNING CREDS** window supply the following information:

A **Bundle ID** to uniquely identify your app on the device and in Google Play Store.

Enter your Android **keystore** credentials or select **Auto Generated Debug Keystore** to use a development **keystore**. For more information on Android keystores, see Android Keystore System.

iOS Credentials

In the IOS **SIGNING CREDS** window supply the following information:

A **Bundle ID** to uniquely identify your app on the device.

The Xcode version with which to build the app.

Enter your IOS credentials. For more information on iOS credentials, see Building for iOS.

Using Git submodules

If your project is using private Git submodules, make sure that the URLs present in your *.gitmodules* file are using the `git@` syntax instead of `https://` or `git://`.

For example:

git@github.com:youraccount/yourrepo.git (for GitHub)

git@bitbucket.org:youraccount/yourrepo.git (for Bitbucket)

git@gitlab.com:youracccount/yourrepo.git (for GitLab)

2018–07–16Page amended with editorial review

Using the Unity Editor to configure Unity Cloud Build for Git

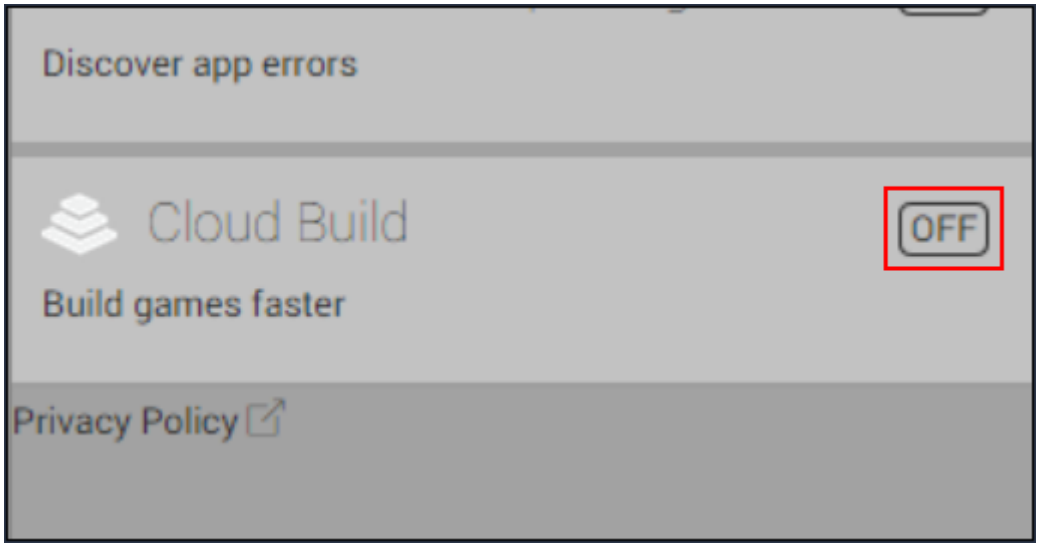
Leave feedback

Unity Cloud Build supports projects stored in Git repositories. Your repository can be hosted on GitHub, GitLab, Bitbucket, or private servers.

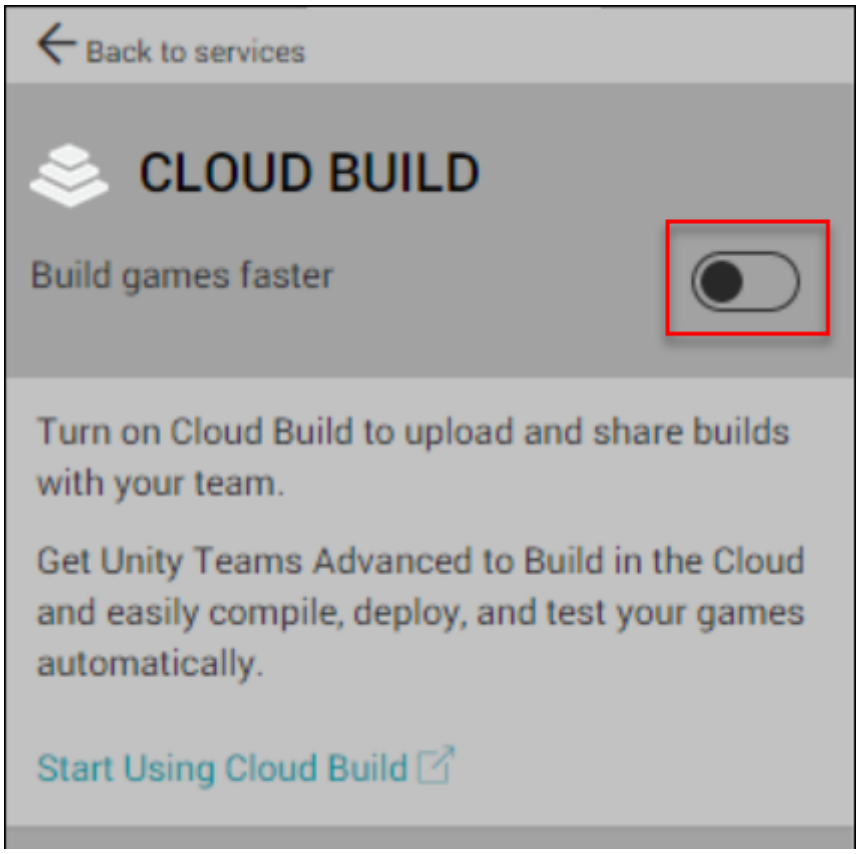
Note: This feature requires a subscription to Unity Teams Advanced. For more information, see the Unity Teams page.

Enabling Cloud Build

In the **Services** window, next to **Cloud Build**, click the **OFF** button to switch it **ON.**



In the **CLOUD BUILD** window, click the **Cloud Build** toggle to enable Cloud Build for your Project.



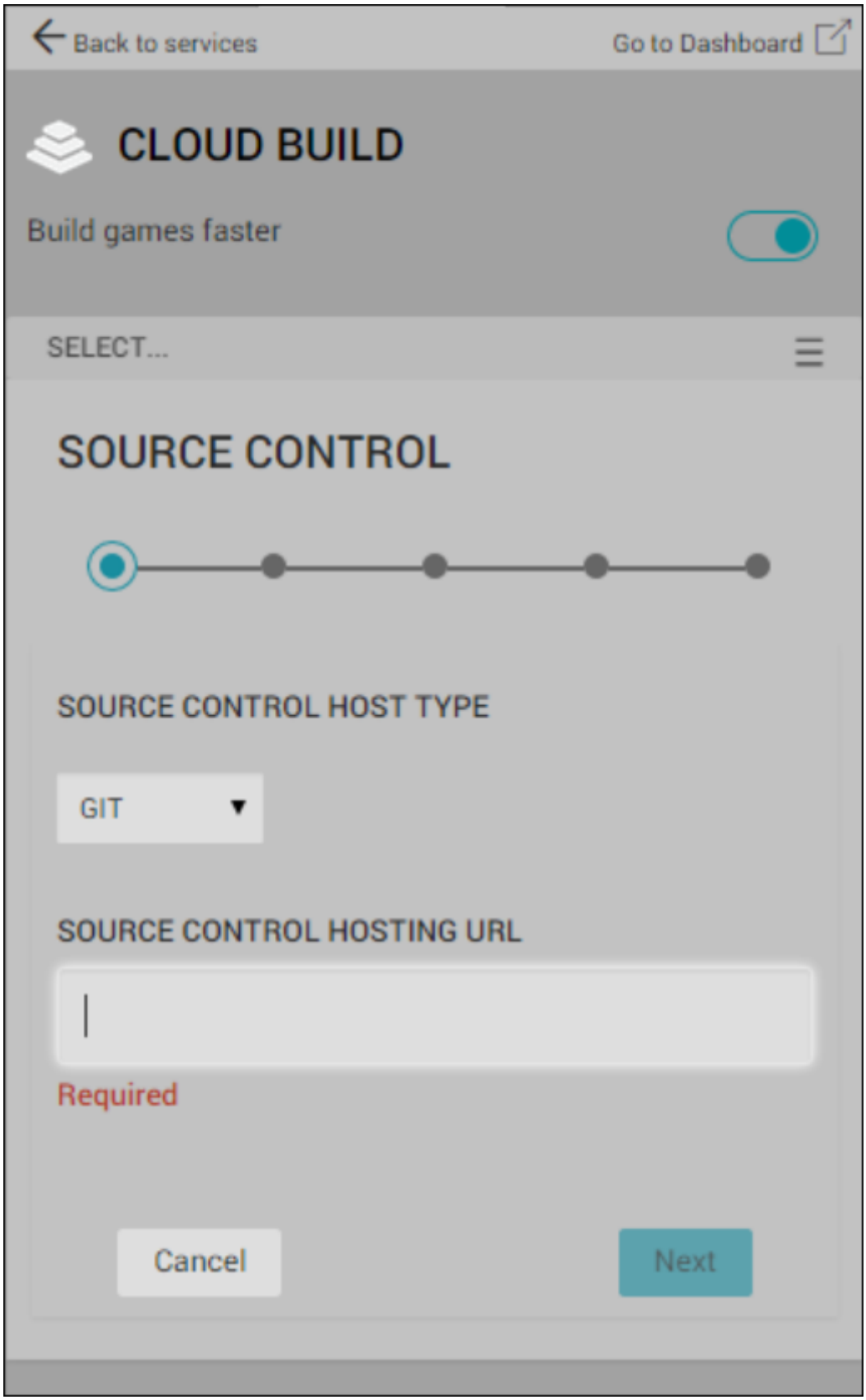
Setting the source control hosting type

In the services **SOURCE CONTROL** window:

From the **SOURCE CONTROL HOST TYPE** drop-down menu, select **GIT**.

In the **SOURCE CONTROL HOSTING URL** field, enter the URL of your Git repository. For information on the format of the URL, see the **URL syntax** section below.

Click **Next**.



URL syntax

To connect to your repository, you must specify the URL to your Git server. You can specify the URL to use the following protocols:

HTTPS

GIT

SSH

The following are examples of URLs for GitHub, bitbucket, and GitLab:

https://github.com/youraccount/yourrepo

git://github.com/youraccount/yourrepo.git

git@bitbucket.org:youraccount/yourrepo.git

git@gitlab.com:youracccount/yourrepo.git

If you are hosting Git on a private server, you must use SSH to connect to your repository.

Use the format that is most convenient for you. Unity Cloud Build automatically re-writes the URL into the format it needs.

When Unity Cloud Build connects to the hosting site, it automatically detects whether your repository is public or private. If your repository is public, Unity Cloud Build automatically connects to it and you can skip to the Target Setup section below.

Using private repositories

If your repository is private, Cloud Build must use SSH to connect to it. When Cloud Build detects that the repository is private, it displays the **Grant us access to your source control** window.

Setting your SSH key on GitHub

To add your SSH key to your project in GitHub:

Sign in to GitHub.

In the upper-right corner of any GitHub page, click your profile photo, then click **Your profile**.

On your profile page, click **Repositories**, then click the name of your repository.

In your repository, click the **Settings** tab.

In the sidebar, click **Deploy keys**, then click the **Add deploy key** button.

In the **Title** textbox, type name to identify this key.

In the **Key** field, paste in your public key from the **Grant us access to your source control** screen, and then click the **Add key** button.

For more information on deploy keys, see Managing Deploy Keys in the GitHub documentation.

You can also add your SSH key to your GitHub account so that all repositories in your account are accessible to Cloud Build. For more information, see Adding a new SSH key to your GitHub account in the GitHub documentation.

Setting your SSH key on Bitbucket

To add your SSH key to your Bitbucket account:

Sign in to Bitbucket.

Click your **avatar** in the lower left of the page.

Click **Bitbucket Settings**.

On the **Settings** page, in the **SECURITY** section, click **SSH keys**.

In **SSH keys**, click the **Add key** button.

In the **Label** field, enter a recognizable name for the key (such as **Unity Cloud Build**.)

Paste the Unity Cloud Build SSH key from the **Grant us access to your source control** screen into the **Key** field.

Click **Add key**.

On the **Grant us access to your source control** screen, click **Next: Target Setup**.

Setting up your target platform

To configure the runtime platform for your app in the Unity Editor, go to the **Services** window and make the following changes:

From the **TARGET LABEL** drop-down menu, select a build platform.

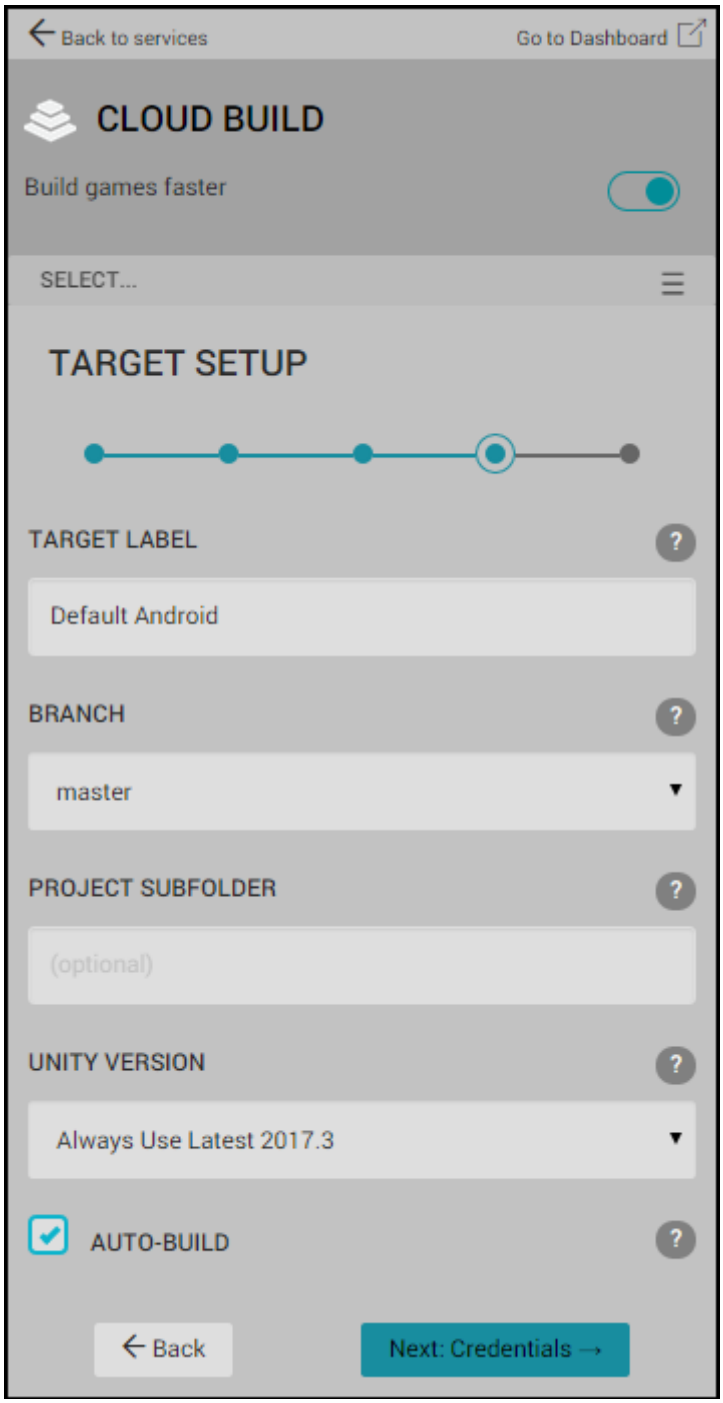
From the **BRANCH** drop-down menu, select the branch from which to build your project. The default branch in most Git repositories is “master”. If you are building for multiple build targets, you can configure a different branch for each build target.

In the **PROJECT SUBFOLDER** field, enter the the folder in your project that contains your Unity project; specifically the **Assets** and **ProjectSettings** folders. The path to the folder is typically similar to the following:

NewGameProject/Src/UnityProject/

If your Unity project is not at the root of your repository, you must complete the **PROJECT SUBFOLDER** field.

Optionally, uncheck the AUTO-BUILD checkbox if you do not want Cloud Build to automatically build your Project when you make changes to your Project. For more information, see Automated Build Generation.



Git submodules

If your project is using private Git submodules, make sure that the URLs present in your *.gitmodules* file are using the `git@` syntax instead of `https://` or `git://`.

For example:

git@github.com:youraccount/yourrepo.git (for GitHub)

git@bitbucket.org:youraccount/yourrepo.git (for Bitbucket)

git@gitlab.com:youracccount/yourrepo.git (for GitLab)

2018–04–10 Page published with editorial review

Using the Unity Developer Dashboard to configure Unity Cloud Build for Mercurial

Leave feedback

Unity Cloud Build supports projects stored in Mercurial repositories. Your can host your repository on Bitbucket or on a private server.

To connect to your repository, **Unity Cloud Build** requires that you supply appropriate access credentials. This applies to Mercurial repositories hosted on Bitbucket or elsewhere.

Configuring Mercurial in the Dashboard

Sign in to the Unity Developer Dashboard.
From the **Projects** page, select your Project.
In the left-hand navigation pane, click **Cloud Build** > **Config**.
If this is the first time you have configured Cloud Build, you see a Build History page informing you that **Cloud Build has not been set up for this project**. Click the **Set up Cloud Build** link to continue.
In the **Source Control** window, click the **Manual** tab.
In the **SCM URL** field, enter the URL of your Mercurial server. For information on the format of the URL, see the **URL syntax** section below.
From the **SCM Type** drop-down menu, select **Mercurial**.
Click the **Next Access** button.

URL syntax

To connect to your repository, you must specify the URL to your Mercurial server. You can specify the URL to use the following protocols:

HTTPS

SSH

The following are examples of URLs for bitbucket:

https://github.com/youraccount/yourrepo

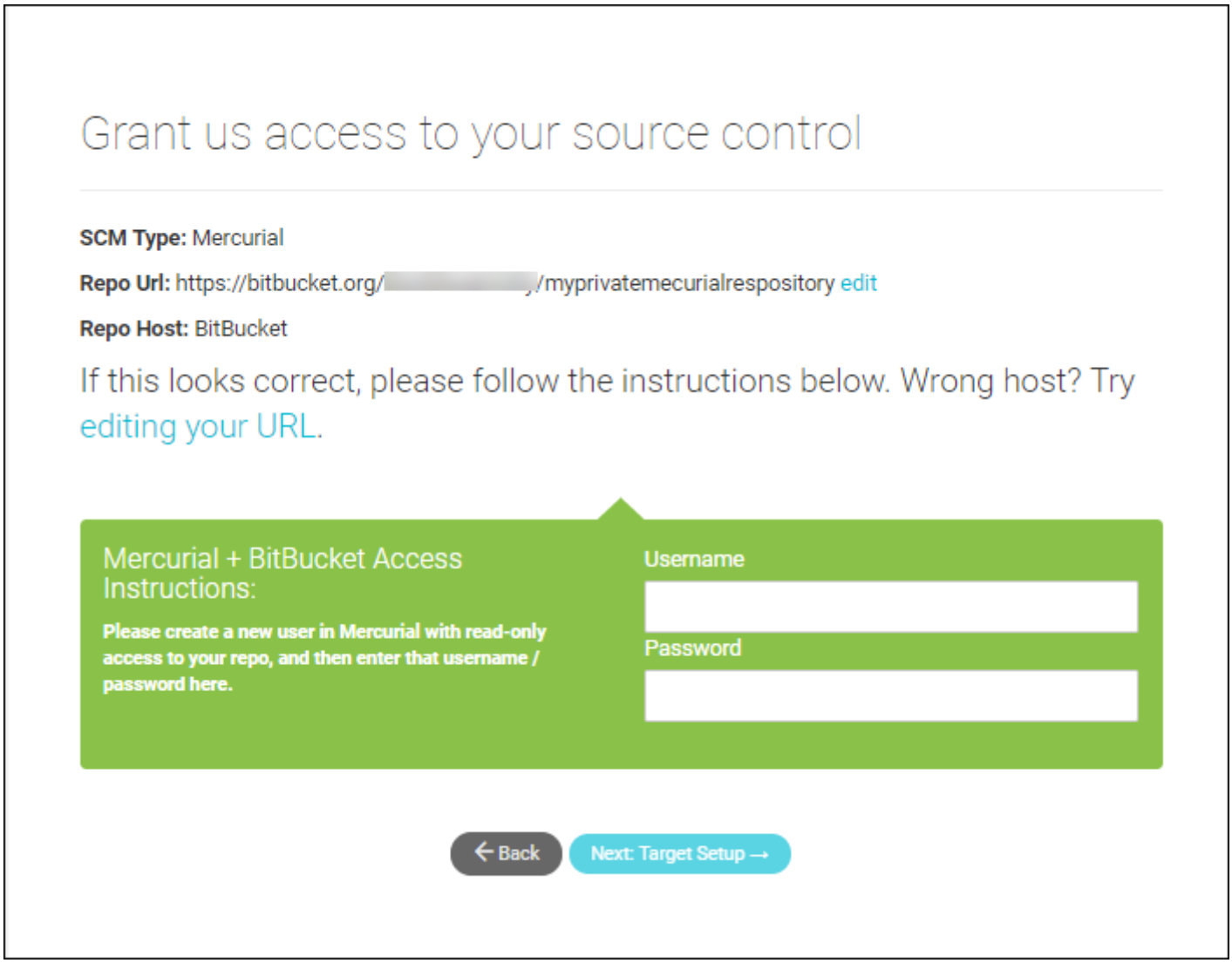
git@bitbucket.org:youraccount/yourrepo.git

Use the format that is most convenient for you. Unity Cloud Build automatically re-writes the URL into the format it needs.

Setting your credentials

On your Mercurial server, create a user name for Unity Cloud Build and assign a secure password. If your Mercurial host supports it, make this a read-only user account.

On the **Grant us access to your source control** window, enter your user name and password, and then click **Next: Target Setup**.



Dashboard grant access window

Setting up the target build platform

In the Target Setup window, select a build platform.

On the Developer Dashboard, on the **NEW BUILD TARGET: BASIC INFO** window:

In the **Target Label** field, enter a name for the build.

If the root folder of your repository doesn't contain your **Assets** and **Project settings**, in the **Project Subfolder** field, enter the URL of the folder that contains your **Assets** and **Project Settings**.

From the **Unity Version** drop-down menu, select the version of Unity with which to build the Project.

If you do not want the Project to automatically build whenever your repository is updated, click the **Auto-build** toggle to disable this feature.

If you are building for the **iOS** or Android platforms, your next step is to enter credentials for the build. Click **Next: Credentials**. For all other platforms, click **Next: Build** to complete the configuration and start the initial build.

Android credentials

On the IOS **SIGNING CREDS** window supply the following information:

A **Bundle ID** to uniquely identify your app on the device and in Google Play Store.

Enter your Android **keystore** credentials or select **Auto Generated Debug Keystore** to use a development **keystore**. For more information on Android keystores, see Android Keystore System.

iOS Credentials

On the IOS **SIGNING CREDS** window supply the following information:

A Bundle ID to uniquely identify your app on the device.

The Xcode version with which to build the app.

Enter your IOS credentials. For more information on iOS credentials, see Building for iOS.

2018–04–10 Page published with editorial review

Using the Unity Editor to configure Unity Cloud Build for Mercurial

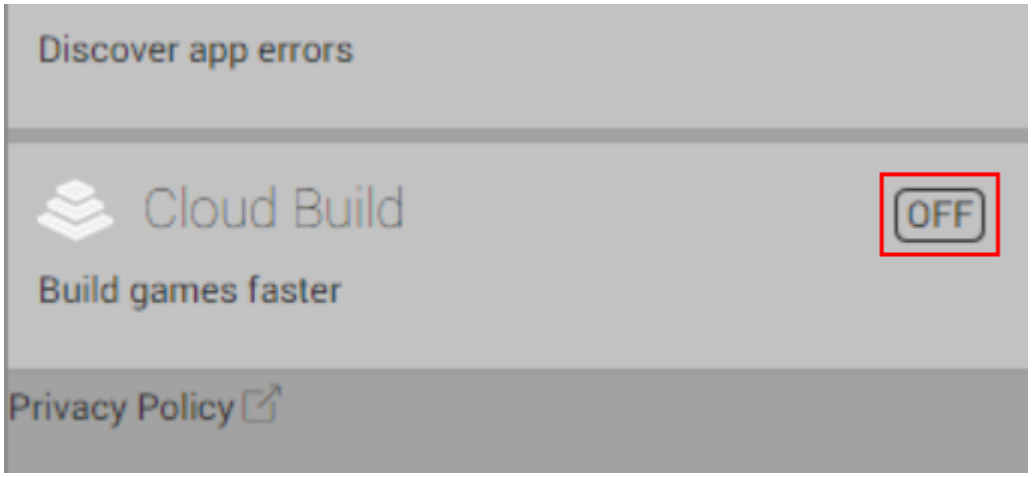
Leave feedback

Unity Cloud Build supports projects stored in Mercurial repositories. Your can host your repository on Bitbucket or on a private server.

To connect to your repository, **Unity Cloud Build** requires that you supply appropriate access credentials. This applies to Mercurial repositories hosted on Bitbucket or elsewhere.

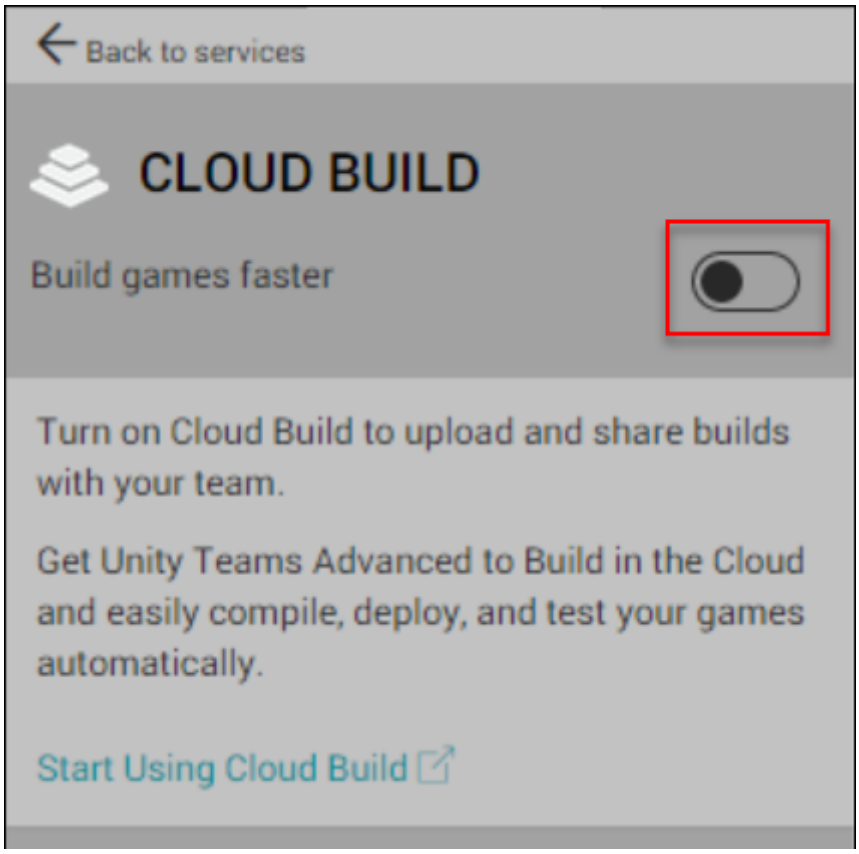
Enabling Cloud Build

In the Services window, next to **Cloud Build click the OFF** button to switch it **ON**:



Cloud build activation state window

In the **CLOUD BUILD** window, click the **Cloud Build** toggle to enable Cloud Build for your Project.



Toggle cloud build window

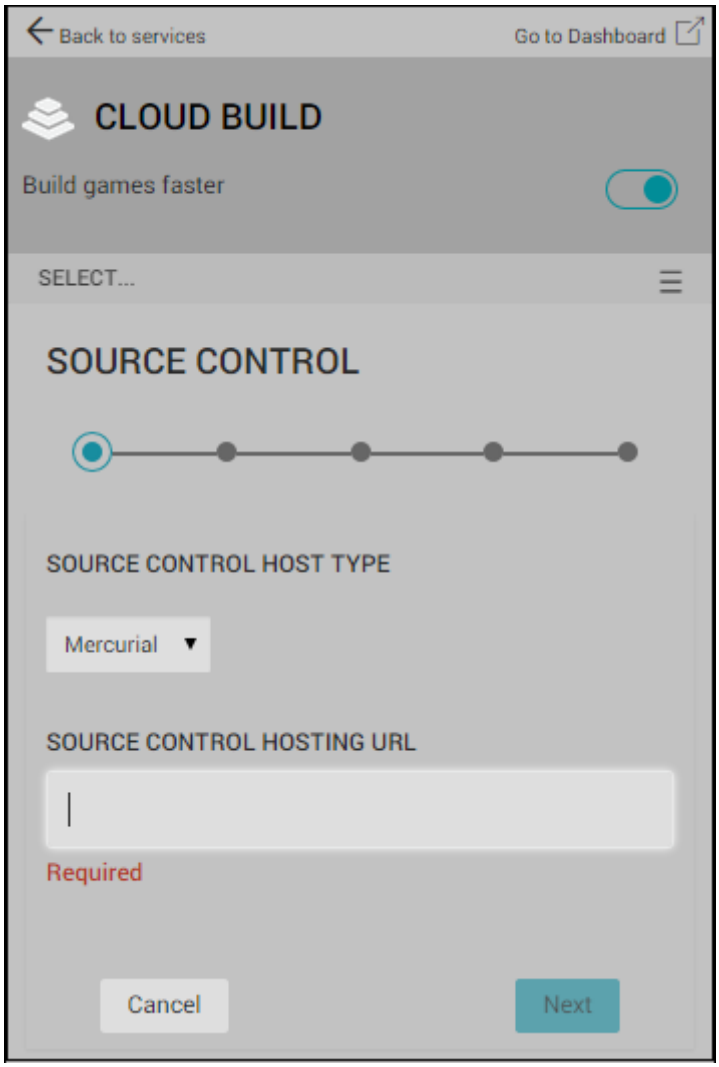
Setting the source control hosting type

In the services__ SOURCE CONTROL__ window:

From the **SOURCE CONTROL HOST TYPE** drop-down menu, select Mercurial.

In the **SOURCE CONTROL HOSTING URL** field, enter the URL of the Mercurial repository. For information on the format of the URL, see the **URL syntax** section below.

Click the **Next** button.



Select source control window

URL syntax

To connect to your repository, you must specify the URL to your Mercurial server. You can specify the URL to use the following protocols:

HTTPS

SSH

The following are examples of URLs for bitbucket:

https://github.com/youraccount/yourrepo

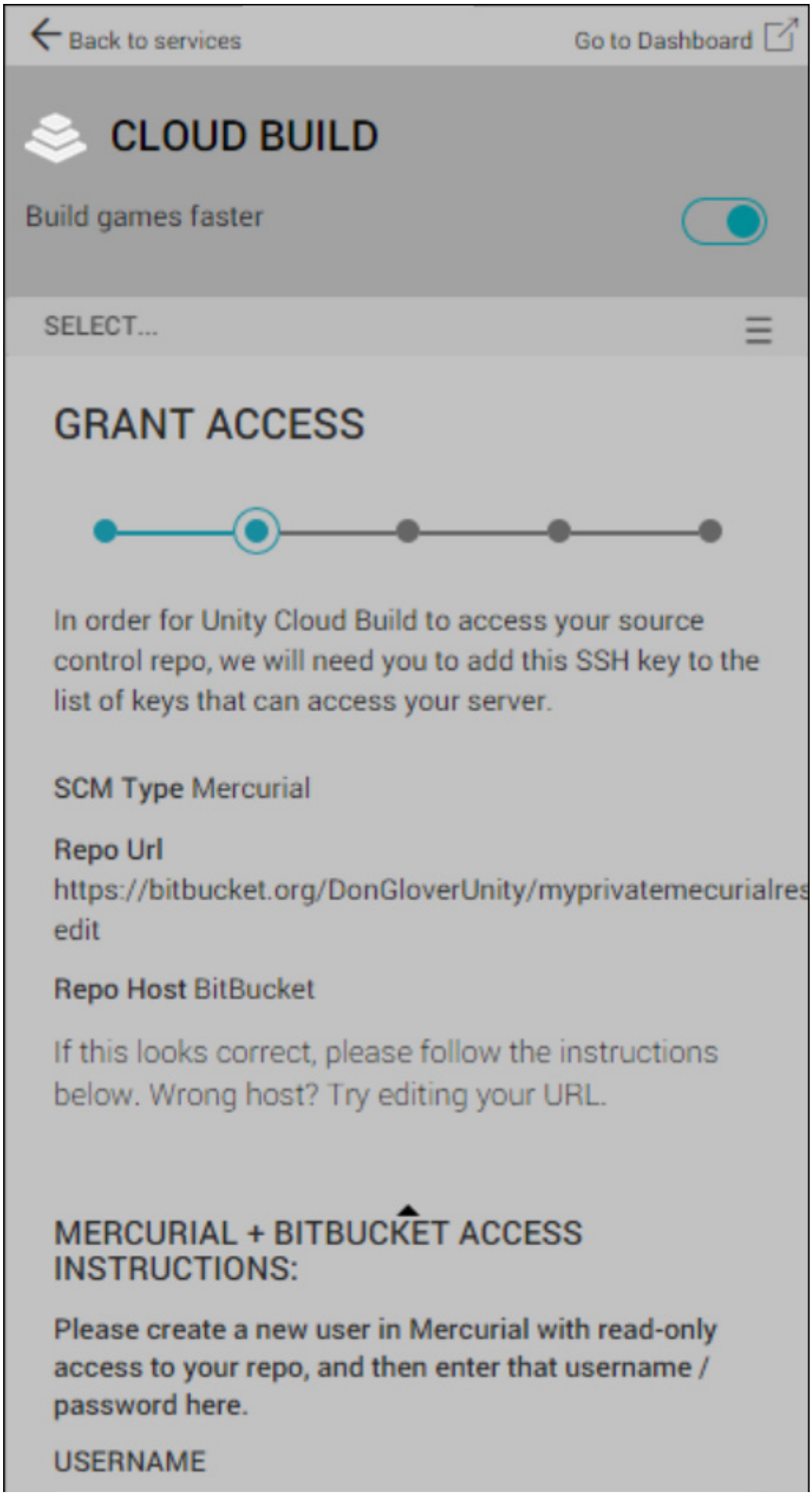
git@bitbucket.org:youraccount/yourrepo.git

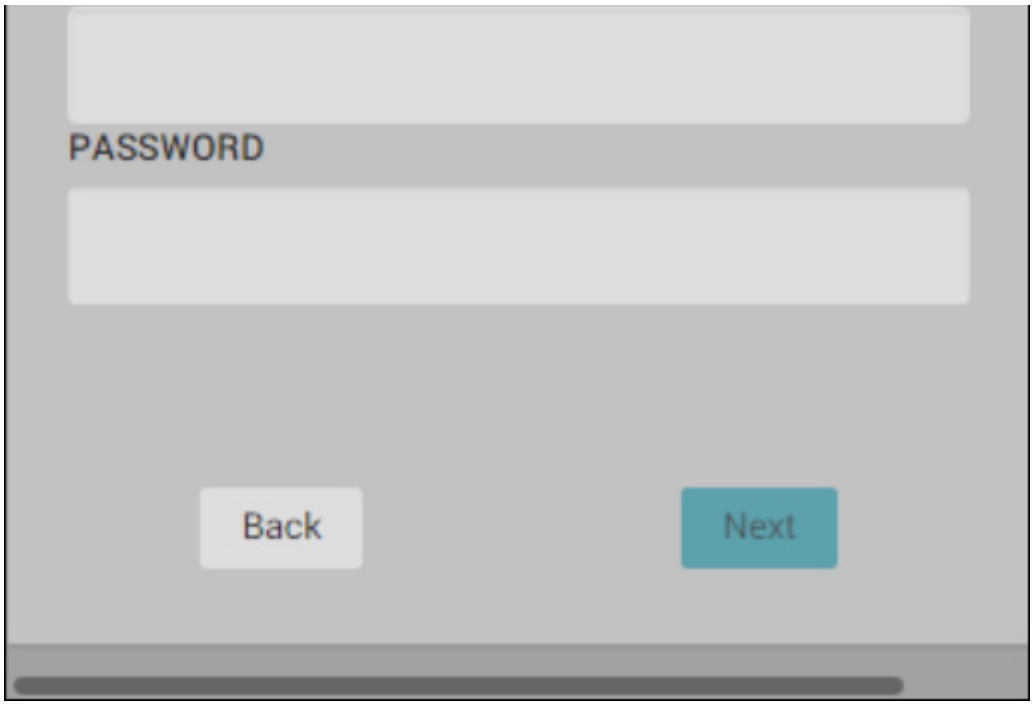
Use the format that is most convenient for you. Unity Cloud Build automatically re-writes the URL into the format it needs.

Setting your credentials

On your Mercurial server, create a user name for Unity Cloud Build and assign a secure password. If your Mercurial host supports it, make this a read-only user account.

In the **GRANT ACCESS** window, enter the user name and password.





Grant access window

Setting up a runtime target platform

To configure the runtime platform for your app, in the Unity Editor, go to the **Services** window and make the following changes:

From the **TARGET LABEL** drop-down menu, select a build platform.

From the **BRANCH** drop-down menu, select the branch from which to build your project. The default branch in most Git repositories is “master”. If you are building for multiple build targets, you can configure a different branch for each build target.

In the **PROJECT SUBFOLDER** field, enter the the folder in your project that contains your Unity project; specifically the **Assets** and **ProjectSettings** folders. The path to the folder is typically similar to *NewGameProject/Src/UnityProject/*

Note: If your Unity project is not at the root of your repository, you must complete the **PROJECT SUBFOLDER** field.

Optionally, uncheck the **AUTO-BUILD** checkbox if you do not want Cloud Build to automatically build your Project when you make changes to your Project. For more information, see Automated Build Generation.

2018–04–10 Page published with editorial review

Using Apache Subversion (SVN) with Unity Cloud Build

Leave feedback

Unity **Cloud Build** supports projects hosted in Apache Subversion (SVN) repositories (see subversion.apache.org).

Username and Password

In your SVN server, create a new user just for **Unity Cloud Build**, along with a secure password. If your SVN host supports it, make this a read-only user account.

Public SSH Keys

Unity Cloud Build does not support connecting to your SVN repository using a public SSH Key. Use a username and password instead.

SSL Certificates

Self-signed SSL certificates are not supported. Unity Cloud Build does not SSL handshake with a server whose certificate is self-signed. The hostname in the certificate must match the hostname being accessed by Unity Cloud Build.

Using the Unity Developer Dashboard to configure Unity Cloud Build for Perforce

Leave feedback

Unity Cloud Build supports Projects stored in Perforce repositories.

Note: This feature requires a subscription to Unity Teams Advanced. For more information, see the Unity Teams page.

This topic covers:

Configuring Perforce in the Dashboard
Setting up a target build platform

Configuring Perforce on the Dashboard

To configure Cloud Build to build your Project from a Perforce repository:

Sign in to the Unity Developer Dashboard.

On the **Projects** page, select your Project.

In the dashboard **Overview** window, on the **Cloud Build** tile, click **OPEN CLOUD BUILD DASHBOARD**.

In the **Build History** window, click **Set up Cloud Build**.

In the **Source Control** window, click the **MANUAL** tab.

In the **SCM URL** field, enter the URL of your Perforce server in either of the following formats:

Non-SSL: host:port
SSL: ssl:host:port

To connect to your repository, you must specify the URL to your Perforce server. For information on the format of the URL, see the **URL syntax** section below.

From the **SCM Types** drop-down menu, select **Perforce**.

Click the **NEXT: ACCESS** button. When **Unity Cloud Build** connects to the hosting site, it automatically detects whether your repository is public or private. If your repository is public, Cloud Build automatically connects to it and you can skip to Setting up a target build platform.

On the the **Grant us access to your source control** window, enter the user **Username** and **Password** for your Perforce repository.

Click **NEXT: TARGET SETUP**.

URL syntax

To connect to your repository, you must specify the URL to your Perforce server. You can specify the URL to use the following protocols:

HTTPS
SSL

The following are examples of valid URLs:

https://127.0.0.1:1667
ssl:127.0.0.1:167

Setting up a target build platform

In the dashboard, on the **NEW BUILD TARGET: BASIC INFO** window:

In the **Target Label** field, enter a name for the build.
From the **Client Workspace** drop-down menu, select the workspace from which to build your Project.
If the root folder of your repository doesn't contain your **Assets** and **Project settings**, in the **Project Subfolder** field, enter the URL of the folder that contains your **Assets** and **Project settings**.
From the **Unity Version** drop-down menu, select the version of Unity with which to build the Project.
If you do not want the Project to automatically build whenever your repository is updated, click the **Auto-build** toggle to disable this feature. If you are building for the **iOS** or Android platforms, your next step is to enter credentials for the build. Click **Next: Credentials**. For all other platforms, click **Next: Build** to complete the configuration and start the initial build.

Android credentials

In the Android **SIGNING CREDS** window supply the following information:

A **Bundle ID** to uniquely identify your app on the device and in Google Play Store.
Enter your Android **keystore** credentials or select **Auto Generated Debug Keystore** to use a development **keystore**. For more information on Android keystores, see Android Keystore System.

iOS Credentials

In the iOS **SIGNING CREDS** window supply the following information:

A **Bundle ID** to uniquely identify your app on the device.
The Xcode version with which to build the app.
Enter your iOS credentials. For more information on iOS credentials, see Building for iOS.

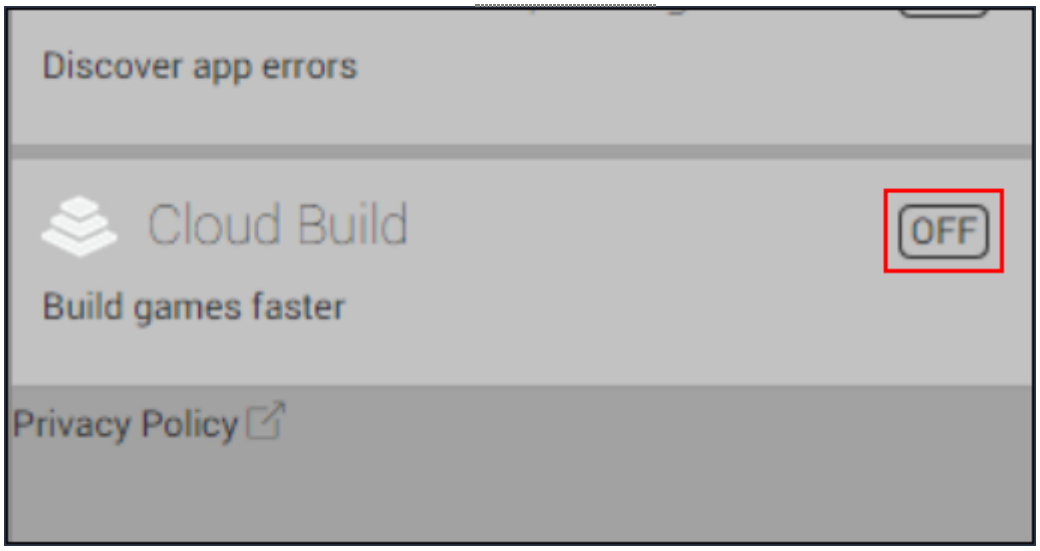
Using the Unity Editor to configure Unity Cloud Build for Perforce

Leave feedback

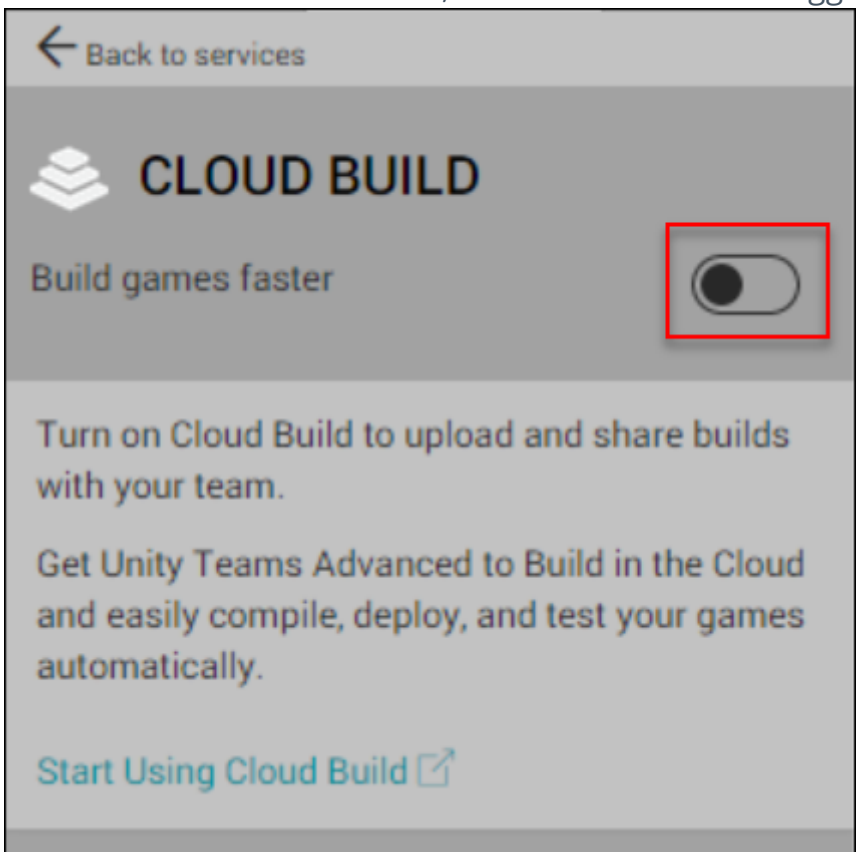
Unity Cloud Build supports Projects stored in Perforce repositories.

Enabling Cloud Build

In the Services window, next to **Cloud Build** click the **OFF** button to switch it **ON**:



In the **CLOUD BUILD** window, click the **Cloud Build** toggle to enable Cloud Build for your Project.

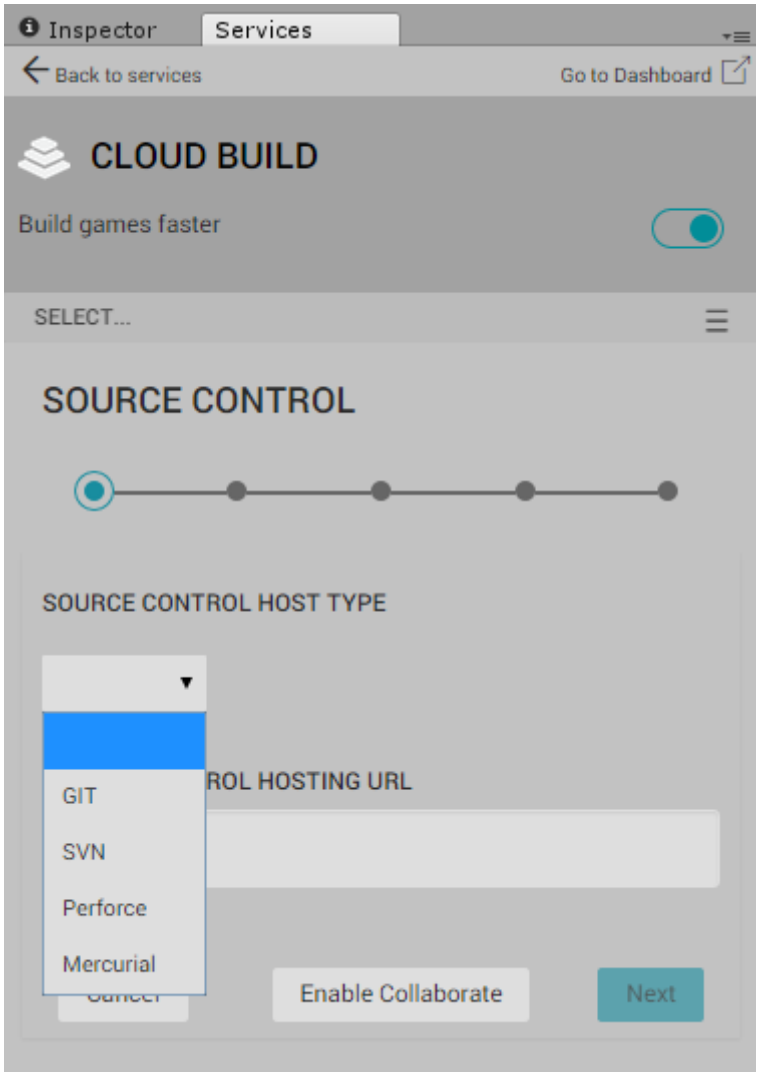


In the **Build History** window, click **Set up Cloud Build**.

Setting the source control hosting type

In the services **SOURCE CONTROL** window:

From the **SOURCE CONTROL HOST TYPE** drop-down menu, select **Perforce**.
In the **SOURCE CONTROL HOSTING URL** field, enter the URL of the Perforce repository. For information on the format of the URL, see the **URL syntax** section below.
Click the **Next** button.



URL syntax

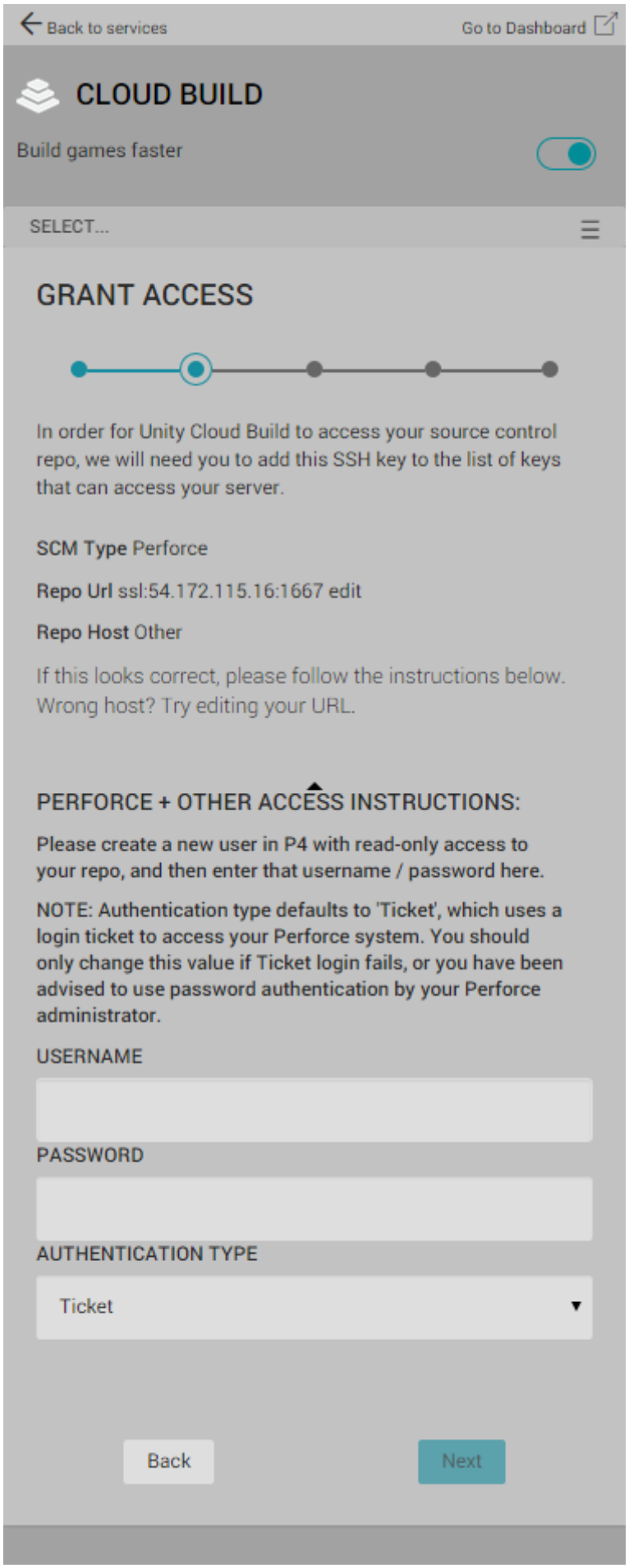
To connect to your repository, you must specify the URL to your Perforce server. You can specify the URL to use the following protocols:

HTTPS
SSL
The following are examples of valid URLs:

https://127.0.0.1:1667
ssl:127.0.0.1:167

Grant access

In the **GRANT ACCESS** window, enter the username and password, then click **Next**.



Setting up a runtime target platform

To configure the runtime platform for your app:

In the **CHOOSE PLATFORM** window, select the build platform from the **PLATFORM** drop-down menu and then click **Next**.
In the **TARGET SETUP** window, from the **CLIENT WORKSPACE** drop-down menu, select the Perforce workspace from which to build your Project.
In the **PROJECT SUBFOLDER** field, enter the folder that contains your Unity Project; specifically the **Assets** and **ProjectSettings** folders. The path to the folder is typically similar to *NewGameProject/Src/UnityProject/.*

Note: If your Unity Project is not at the root of your repository, you must complete the **PROJECT SUBFOLDER** field.

From the **UNITY VERSION** drop-down menu, select the version of Unity to use to build the Project. Optionally, uncheck the **AUTO-BUILD** checkbox if you do not want Cloud Build to automatically build your Project when you make changes to your Project. For more information, see Automated Build Generation.

Building for iOS

Leave feedback

Unity **Cloud Build** helps you automate the process of building your Unity Project for **iOS** devices.

This article describes the prerequisites necessary to build your Project for iOS and how to create the supporting **components** to configure Cloud Build. Topics covered include:

Joining the Apple Developer Program
Creating an iOS certificate and and p12 file
Adding devices to your profile
Creating a provisioning profile
Configuring your app to build for iOS

Joining the Apple Developer Program

To develop iOS apps, you must be a member of the iOS Developer Program. The cost is currently $99 per year, and allows you to build, test, and eventually release your apps in the Apple App Store.

Note: You must have an Intel-based Mac running OS X Yosemite (v10.10) or later to develop and distribute iOS apps and Mac apps.

To join the iOS Developer program:

Log in to the Apple Developer Program page.

Important: Use the Safari browser. You might encounter issues if you use the Chrome or Firefox browsers.

Click the **Enroll** button.

Read the information on the **What You Need To Enroll** page and then click the **Start Your Enrollment** button.

Log in with your Apple ID, or create a new Apple ID if you don't have one

Choose whether to enroll for an Individual/Sole Proprietor/Single Person Business or as a Company/Organization
Individual/Sole Proprietor/Single Person Business: Select this option if you are an individual or sole proprietor/single person company.
Company/Organization: Select this option if you are a company, non-profit organization, **joint** venture, partnership, or government organization.
Enter your contact information and any requested business information.

Read the license agreement. If you agree to the terms, check the acceptance checkbox and click the **Continue** button.

Purchase the program

Note: You cannot access the Apple Developer Program until your membership has been approved. Apple typically takes around 24 hours to approve your membership.

Activate the Program

After you sign in to the Apple Program Developer portal, you'll see a list labeled **Program Resources** on the left. **Click Certificates, IDs & Profiles** to manage the certificates, identifiers, profiles, and devices you need to develop and distribute apps.

Provisioning profiles

A provisioning profile ties developers and devices to an authorized Development Team and enables you to use a device for testing. You must install a *Development Provisioning Profile* on each device on which you plan to run your application code.

Each Development Provisioning Profile contains a set of *Development Certificates, Unique Device Identifiers (UDID)* and an App ID.

To use a device for testing, you must also include your Development Certificate in the profile. A single device can contain multiple provisioning profiles.

Components of a provisioning profile

Certificates determine whether your app is development-only or a release candidate for the App Store. You should use an Ad Hoc Production Certificate so that you can test all the features of your game (like GameCenter, etc).

Identifiers are the unique IDs that you use to identify your project. For basic projects, or if this is your first iOS project, you'll probably want to make an App ID. This is often the same as your Unity3D Project's Bundle ID.

Tip: For more information on signing identities and certificates, see Maintaining Your Signing Identities and Certificates on the Apple developer website.

Devices are the hardware, such as an iPhone, iPad, or iPod on which you plan to test your project. You must retrieve the UDID for each device on which you plan to test your game. You then add the UDID to the Devices section in the iOS developer portal.

Note: Each year, you're allowed to register a fixed number of devices. The maximum number of devices you can register is 100 devices per product family per membership year. For more information see, Registering Devices Using Your Developer Account in the **Maintaining Identifiers, Devices, and Profiles** topic on the Apple developer website.

Creating an iOS certificate and p12 file

When you create a certificate, you must decide whether to create a *Development Certificate* (used only for testing), or a *Production Certificate*, which you use to distribute your app via the App Store.

Tip: Create a Production Certificate. While either certificate type works for development, using a Production certificate makes the process of releasing your app to the Apple store easier.

Create a certificate

Sign in to your Apple Developer Program.
Click **Member Center** > **Certificates** > **Identifiers & Profiles** > **Certificates**.
In the left-hand column, under **Certificates**, click **All**.
In the screen labeled **What type of certificate do you need**, choose the type of certificate to generate. Typically, if you're just starting off, you want to pick an **App Store and Ad Hoc certificate production certificate**.
Next, use the *Keychain Access* program on your Mac (open up your Finder and find it in *Applications/Utilities*) to generate a Certificate Signing Request (CSR) file. Follow the instructions in the iOS portal to complete this step.
Make a note of where you save the CSR file, as you'll need it for the next step.
From the **Generate your certificate** screen, upload your CSR file (which might have the extension *.certSigningRequest*). Click the **Choose File** button and select your CSR file, and then click the **Generate** button.
To download your certificate to your Mac, click the **Download** button on the **Your Certificate is Ready** screen.
Make sure to store this file somewhere safe, and back it up.
To add the certificate to a keychain, locate the certificate file and double-click it. This opens the Keychain Access program. If you get a popup with the message "Do you want to add the certificate to a keychain?", choose login and click the **Add** button.

Export a p12 file

To create apps using **Unity Cloud Build**, you must convert your certificate file to a p12 file. A p12 file is a file that contains your private key and certificate and is used to sign your code. Typically, if you are developing a project in native Xcode, this process is handled for you behind the **scenes**.

To generate a p12 file:

On your Mac, open up your Finder and in Applications/Utilities, open the Keychain Access program.

In the left-hand column, under keychains verify that **Login** is selected.

In the left-hand column, under **Category** verify that **My Certificates** is selected. In the main **Keychain Access** pane, select your certificate.

Note: Typically your certificate is located under **My Certificates**. If you do not find it there, check under **Certificates**.

From the **File menu**, select **File** > **Export Items**, or right-click and select **Export**.

Select Personal Information Exchange (.p12) from the File Format drop-down menu.

Note: If **Login** under **Keychains and My Certificates** under **Category** is not selected, the p12 option is greyed out.

You are asked to create a password for the p12 file.

Important: Record the password somewhere safe. You must supply the password when setting up iOS builds on Unity Cloud Build.

Adding devices

Apple requires the UDID for each device on which you intend to install your app. This is only required for development purposes. Once your app is accepted into the App Store, it is available for anyone to download and install; provided they have the correct version of iOS and meet any other necessary requirements.

Finding your UDIDs

You can use iTunes to retrieve the UDID of your device. For a thorough walkthrough of the retrieval process, see WhatsMyUDID.com.

The basic steps are:

Open *iTunes* on your Mac.
Connect the device (iPhone, iPad, etc) to the computer.
In *iTunes*, select the device.
You should see a screen with the device name, capacity, and other details about the device. To display the UDID, click the **Serial #** field.
Copy and paste the UDID into a document where you can retrieve it later.
Close *iTunes* and disconnect your device.

Add the UDID in the Apple developer portal

Now that you have your device UDID(s), you're ready to add them to the Apple Developer Portal:

Click the **All** section under **Devices** in the left column of the iOS Developer Portal.
To add a new UDID, click the Add button (**+**) in the upper-right corner
Give the device a name you will recognize, and copy-paste the UDID you got from *iTunes* into the UDID field.
Click **Continue**.
Repeat the steps for each of your devices.

Create App ID

Now that you've created your iOS Certificate, you can create an App ID which is used to create an App ID:

In the left-hand column of the Apple Developer Portal, click **App ID**.

In the **Register iOS App IDs** pane, at the top right, click the Add button (**+**).

In the **Registering an App ID** pane, enter the following information:
App ID Description: The name of your app, without any special characters.
App ID Suffix: If you intend to incorporate specific services such as Game Center or In-App Purchases, create an Explicit ID. If you don't need these services, then create a **Wildcard App ID**. A Wildcard App ID allows you to re-use the App ID for multiple projects.
App Services: Optional. Indicate whether you plan on using any of the App Services provided by Apple.
For more information on registering an App ID, see Maintaining Identifiers, Devices, and Profiles.

Click the **Continue** button.

On the **Confirm your App ID** page, check the information you've provided and then click the **Submit** button.

Create a provisioning profile

The next step is to generate a *.mobileprovision* file. The *.mobileprovision* file combines your p12 certificate, App ID, and identifies the UDIDs of the devices on which you are testing your app.

In the Apple Developer portal, click **Certificates, IDs & Profiles**.
In the left-hand column of the Apple Developer Portal, under **Provisioning Profiles**, select **All**.
To add a new Provisioning Profile, click the Add button (**+**) in the upper-right corner.
Under **Development**, select the type of provisioning profile to create and click **Continue**. **Note**: If you're just getting started, you should use a **Distribution** > **Ad Hoc certificate**, as this allows you to build your game and test it on devices.
Select the App ID you want to use for development, and click the **Continue** button.
Select one or more development certificates, and click the **Continue** button.
Select one or more devices, and click the **Continue** button.
Enter a profile name, and click the **Generate** button.
Click the **Done** button.
Download the generated *.mobileprovision* file to your desktop machine.

Configuring your app to build for iOS

To configure your iOS Cloud Build, you need the following items:

Your provisioning profile (.mobileprovision)
Your .p12 file
The password that you used to protect your .p12 file
For basic iOS usage this process should be sufficient. For projects that include Xcode frameworks, you must perform some additional configuration.

Using Xcode Frameworks

To add Xcode frameworks manually, use the Xcode Manipulation API. The API is maintained by the Unity iOS team and API allows you to manage external Xcode frameworks.

For an example of a Unity Project that uses the API, see the UpdateXcodeProject example Project on BitBucket. You can use the example to experiment and to learn from.

One of the **plugins** of the example Project is an external Xcode project manipulation DLL. The DLL is the build product of the source available in Unity's Bitbucket repository. A preferred way to include Xcode project manipulation functionality is to copy the C# source code files to the **Assets**/Editor folder in your Project.

There are two ways you can use the Xcode Manipulation API:

Use the built-in Unity PostProcessBuildAttribute, which is executed before the Unity Cloud Build Post-Export method runs.
Use the Unity Cloud Build Post-Export method (requires access to advanced settings).

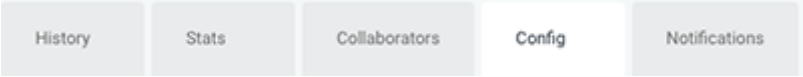
Advanced options

Leave feedback

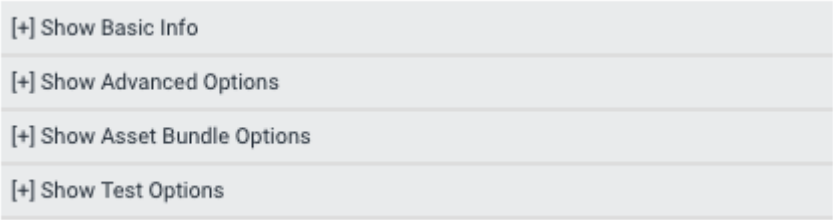
Once you've configured your project in Unity **Cloud Build**, you can set up advanced build options for each of your build targets.

These options are designed to accommodate the more complex build processing options supported in the Unity Editor.

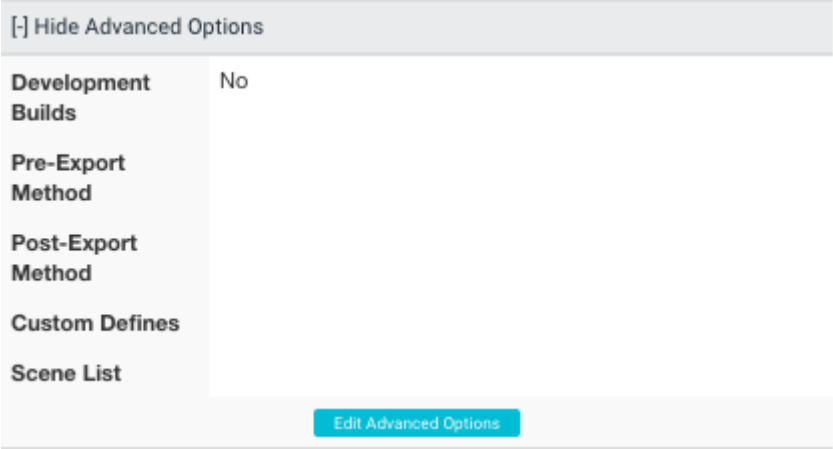
To access your build target's advanced options, go to the Unity Developer website. Select your project, enter the **Unity Cloud Build** section for the project, then select the **Config** tab, shown below.



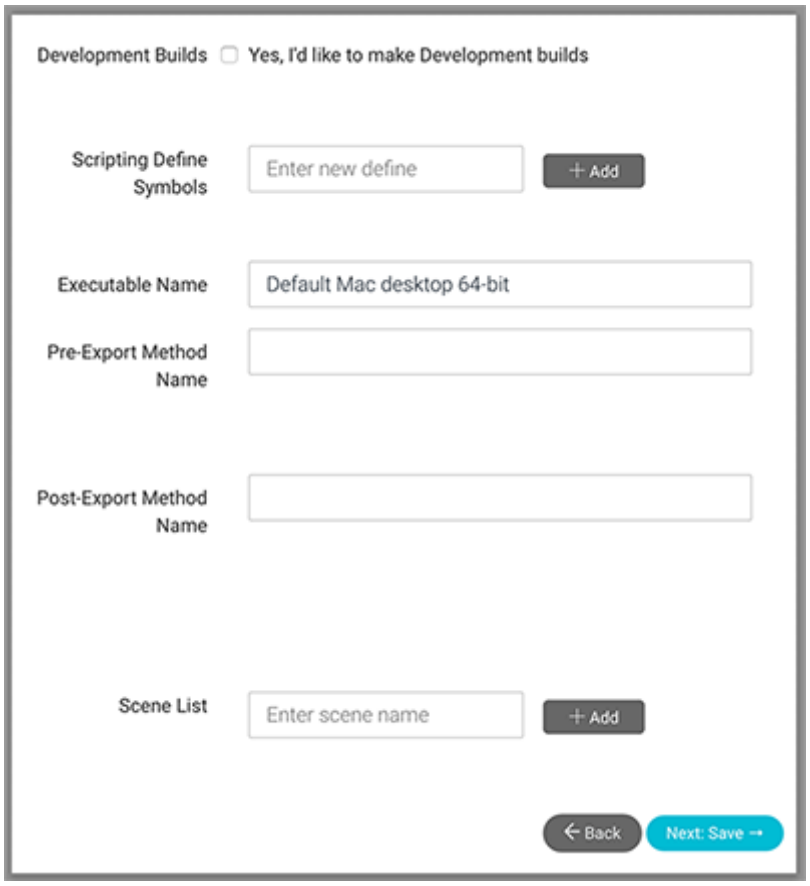
Click **[+] Show Advanced Options**, shown below. This displays a drop-down list of the build target's **Advanced Options**.



The title changes to **[-] Hide Advanced Options** when the **Advanced Options** drop-down is expanded.



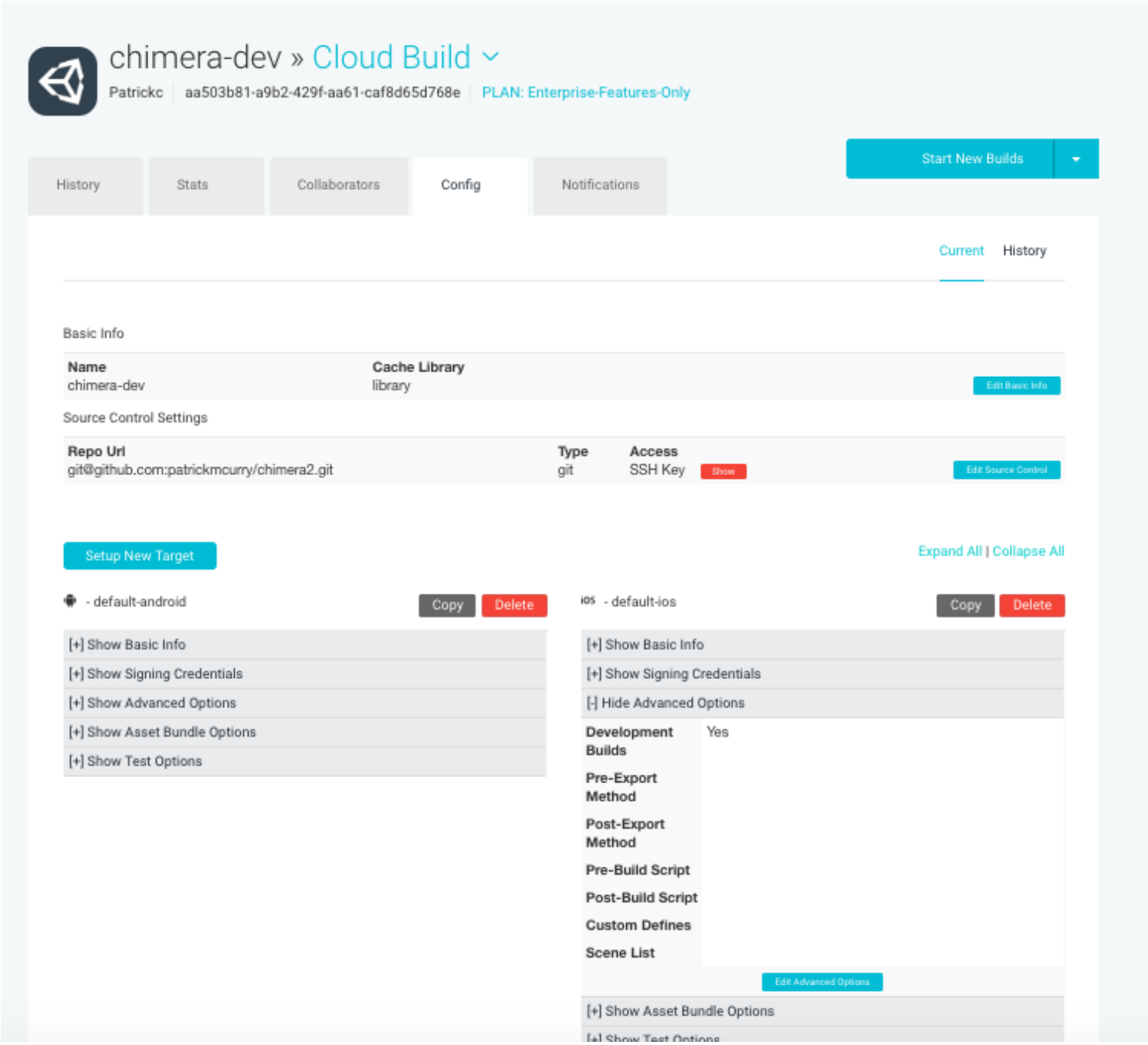
Click **Edit Advanced Options** to bring up a screen to configure these options.



The Edit Advanced Options screen

Advanced options per build target

All advanced options are set per build target. This means that, for example, when you click the **Advanced Options** link for an **iOS** target, those options are only for that **iOS** target. When you click **Advanced Options** for an Android target, those options are only for that Android target. This allows you to use different pre- and post-methods per platform, per build target.



Making development builds

A development build includes debug symbols and enables the **Profiler**.

See documentation on Development builds for more information.

Pre- and post-export methods

Sometimes you need to manipulate the project files before or after the project is built in Unity. Examples include copying variables from an external file into the project, processing **Assets**, or working with plug-ins that require special treatment.

See documentation on Pre- and post-export methods for more information.

Using Xcode frameworks

When building for iOS, you might need to include various frameworks after the Unity build process is done, but before the Xcode build process starts.

See documentation on Xcode frameworks for more information.

Custom scripting #define directives

Unity includes a feature named **Platform Dependent Compilation**. This consists of preprocessor directives that let you partition your **scripts** to compile and execute a section of code exclusively for one of the supported platforms. You can also specify your own #define directives for each build target.

See documentation on Custom scripting #define directives for more information.

Custom Scene list

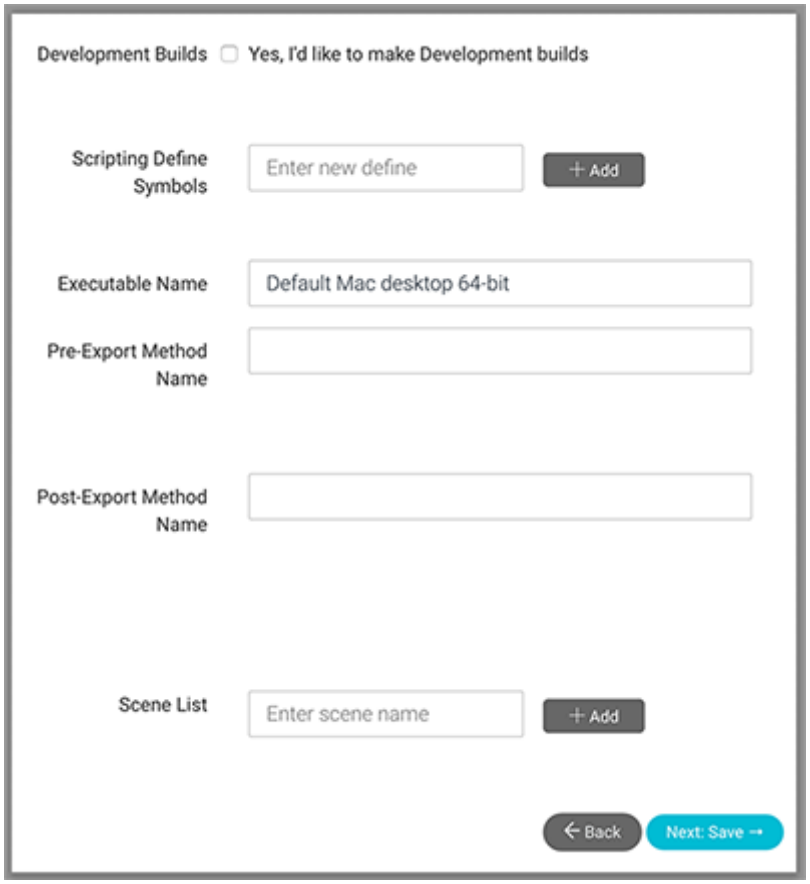
Use this if you want to configure a build target to build a set of **Scenes** that's different to what is set up in the project's **Build Settings** menu in the Unity Editor.

See documentation on Including specific Scenes for more information.

Development builds

Leave feedback

Unity **Cloud Build** builds your projects for release by default. To create a development build, go to the build target's **Advanced Options**. (See documentation on accessing and editing Advanced Options.)



The Edit Advanced Options screen

Check the **Development Builds** box. When enabled, this sets two flags in the `BuildOptions` API:

`Development`: A development build includes debug symbols and enables the **Profiler**.

`AllowDebugging`: Allow script debuggers to attach to the player remotely.

Note that to profile a web player build, you need to have the development version of the web player installed. You can only profile the .unity3d file when the development web player build is active. See documentation on the Profiler window for more information on profiling your development build.

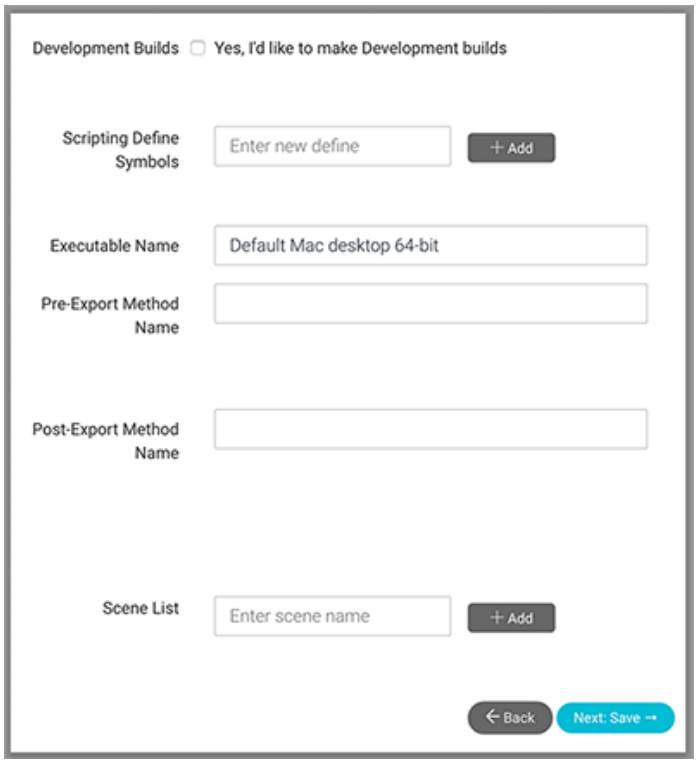
Pre- and post-export methods

Leave feedback

The pre- and post-export methods allow you to trigger actions before and after your Unity Project is built. These methods must exist as code within your Project in the Editor folder in your Asset folder. If this folder doesn't exist in your directory, right click within your Asset folder and go to **Create__>__Folder** and name it "Editor".

Important: The UnityEngine.CloudBuild.BuildManifestObject class is only available when running in **Cloud Build** (as in, not locally). To compile your code locally, wrap your pre- and post-export methods in an `#if UNITY_CLOUD_BUILD` block.

You set pre- and post-export methods in the Advanced Options of the build target. For more information, see Advanced Options.



The Edit Advanced Options screen

Pre-export method name

To use a pre-export method, create a public static method in your Unity Project that contains the code which you want executed before the Project is exported by the Unity Editor.

```
public static void PreExport()
```

You can enable Cloud Build to pass the build manifest of the current build to the pre-export method by specifying a BuildManifestObject object as a parameter in the method signature. You can then make changes to the project or player settings before the Project is exported.

```
public static void PreExport(UnityEngine.CloudBuild.BuildManifestObject manifest)
```

When **Unity Cloud Build** calls the method, it passes a BuildManifestObject object as an optional parameter, where BuildManifestObject is the build manifest of the current build.

For more information, see Build manifest as ScriptableObject.

Post-export method name

To use the post-export method, create a public static method in your Unity Project that contains the code which you want executed after the Project is exported by the Unity Editor.

```
public static void PostExport(string exportPath)
```

When Unity Cloud Build calls the method, it passes a string:

For non iOS build targets, the string contains the path to the exported Project.

For iOS Projects, the string contains the path to the exported Xcode project. You can use the path to locate the exported Xcode project to perform additional preprocessing before Xcode is called to complete the build process.

Note: If you've tagged any methods in your code with the Unity PostProcessBuildAttribute, those methods are executed before any methods configured as post-export methods in Unity Cloud Build.

Xcode frameworks

Leave feedback

If your **iOS** project requires additional Xcode frameworks, use the PBXProject API to add those frameworks to the Xcode project files created by **Unity Cloud Build**.

There are two ways to call this API to manipulate the Xcode project:

Use the built-in Unity PostProcessBuildAttribute, which is executed before the Unity **Cloud Build** Post-Export method runs.

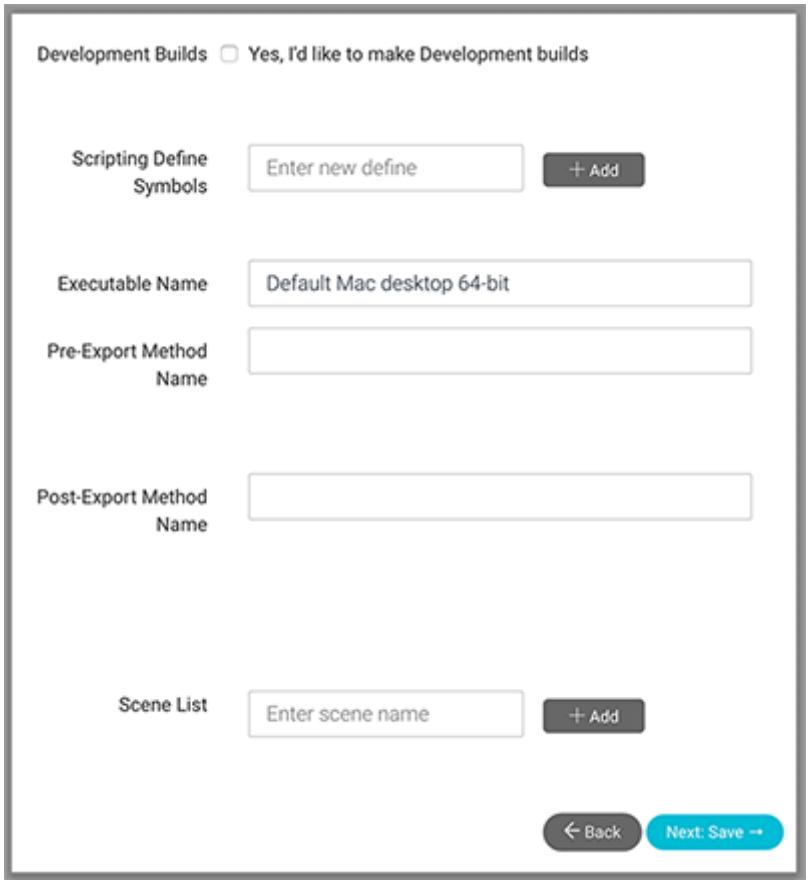
Use the Unity Cloud Build post-export method.

Note: if you are using a version of Unity earlier than 5.0, add this functionality to your Unity project using the Xcode Manipulation API provided by Unity on Bitbucket.

Custom scripting #define directives

Leave feedback

Using **Unity Cloud Build**, you can create custom scripting #define directives. On the Unity Developer website, go to the build target's **Advanced Options**. (See documentation on accessing and editing Advanced Options.)



The Edit Advanced Options screen

In the **Scripting Define Symbols** field, you can add your own custom scripting #define directives to the the built-in selection available. For each build target, enter the names of the symbols you want to define. You can then use these symbols as the conditions in #if directives, just like the built-in ones.

For more information, see documentation on Platform-dependent compilation.

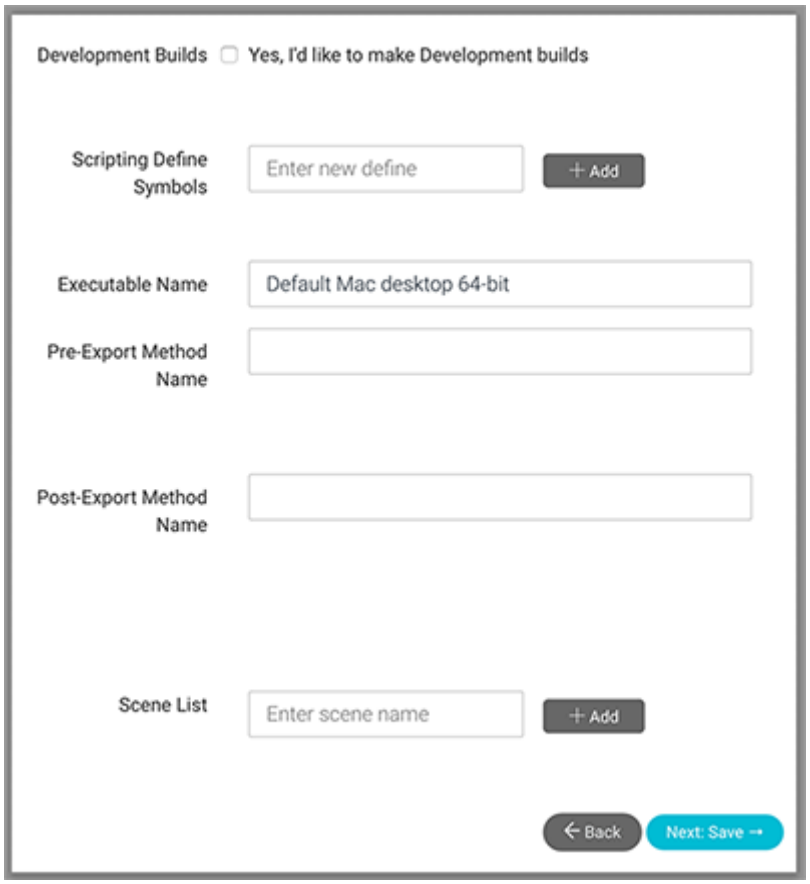
Including specific Scenes

Leave feedback

By default, Unity Cloud Build builds the Scenes you've added to your project in the Unity Editor to **File** > **Build Settings** > **Scenes in Build**. Click **Add Open Scenes** to add additional Scenes. See documentation on Publishing builds for more information.

To instruct **Unity Cloud Build** to ignore the **Scenes in Build** list in the Unity Editor and build a different list of **Scenes**, you need to provide a **Scene List** via the Unity Developer website.

On the Unity Developer website, go to the build target's **Advanced Options**. (See documentation on accessing and editing Advanced Options.)



The Edit Advanced Options screen

Add the list of Scenes you want to include to the **Scene List**. Provide the file path as a relative path from the **Assets** directory to each Scene file you wish to include.

This is useful for development. For example, you may have one build target that includes all the Scenes, but takes a long time to build. If you're only working on one Scene, you might find it helpful to create another build target that only includes that Scene. This should build much faster, and allow for more rapid iteration of development versions.

Build manifest

Leave feedback

It's often useful for your game's run-time code to know key information about the build itself. Information like the name and number of the build is very useful when reporting bugs or tracking **analytics**. To help facilitate this, **Unity Cloud Build** injects a "manifest" into your game at build time, so that this key data can be accessed later at runtime.

The Unity **Cloud Build** manifest is provided as a JSON formatted TextAsset. This is stored as a game resource, accessible via `Resources.Load()`. The build manifest contains the following values:

Value:	Properties:
`scmCommitId`	The commit or changelist that was built.
`scmBranch`	The name of the branch that was built.
`buildNumber`	The Unity Cloud Build "build number" corresponding to this build.
`buildStartTime`	The UTC timestamp when the build process was started.
`projectId`	The Unity project identifier.
`bundleId`	The `bundleIdentifier` configured in Unity Cloud Build (iOS and Android only).
`unityVersion`	The version of Unity that Unity Cloud Build used to create the build.
`xcodeVersion`	The version of XCode used to build the project (iOS only).
`cloudBuildTargetName`	The name of the build target that was built.

The manifest TextAsset, called **UnityCloudBuildManifest.json**, is written to the *Assets/UnityCloud/Resources* folder.

For local testing

To test the build manifest functionality locally, name your file **UnityCloudBuildManifest.json.txt** (but don't commit this file to your project's **Assets/UnityCloud/Resources** folder in your code repository, because it might interfere with the Unity Cloud Build manifest file).

Using the manifest

You can access the manifest at runtime via:

Manifest as JSON

Manifest as ScriptableObject

Build manifest as JSON

Leave feedback

You can access the Unity Cloud Build manifest as JSON at runtime. This **Asset** requires either custom parsing logic or the use of a third party JSON parser.

The following example C# code demonstrates loading and parsing the build manifest with the MiniJSON parser on GitHub Gist (gist.github.com/darktable/1411710):

```
using UnityEngine;
using System.Collections.Generic;
using MiniJSON;

public class MyGameObject: MonoBehavior
{
    void Start()
    {
        var manifest = (TextAsset) Resources.Load("UnityCloudBuildManifest.json"
        if (manifest != null)
        {
            var manifestDict = Json.Deserialize(manifest.text) as Dictionary<str
            foreach (var kvp in manifestDict)
            {
                // Be sure to check for null values!
                var value = (kvp.Value != null) ? kvp.Value.ToString() : "";
                Debug.Log(string.Format("Key: {0}, Value: {1}", kvp.Key, value))
            }
        }
    }
}
```

Build manifest as ScriptableObject

Leave feedback

`BuildManifestObject` is a ScriptableObject that you can use to access the values in the Build manifest via script, without needing to manually load the **UnityCloudBuildManifest.json** TextAsset.

It is an optional parameter to the pre-export invoked by Unity Cloud Build, if the **UnityCloudBuildManifest.json** TextAsset has not been written. (See documentation on Manifest as JSON.)

The following example C# code demonstrates a pre-export method that updates the `bundleVersion` in `PlayerSettings` based on the `buildNumber` provided in the manifest. For more info about pre-export methods, see documentation on Pre- and Post-export methods.

```
using UnityEngine;
using UnityEditor;
using System;

public class CloudBuildHelper : MonoBehaviour
{
    public static void PreExport(UnityEngine.CloudBuild.BuildManifestObject mani
    {
        PlayerSettings.bundleVersion = String.Format("1.0.{0}", manifest.GetValu
    }
}
```

This is the public interface for the `BuildManifestObject` class:

```
namespace UnityEngine.CloudBuild
{
    public class BuildManifestObject : ScriptableObject
    {
        // Try to get a manifest value - returns true if key was found and could
        public bool TryGetValue<T>(string key, out T result);
        // Retrieve a manifest value or throw an exception if the given key isn'
        public T GetValue<T>(string key);
        // Set the value for a given key.
        public void SetValue(string key, object value);
        // Copy values from a dictionary. ToString() will be called on dictionar
        public void SetValues(Dictionary<string, object> sourceDict);
        // Remove all key/value pairs.
        public void ClearValues();
        // Return a dictionary that represents the current BuildManifestObject.
        public Dictionary<string, object> ToDictionary();
```

```
        // Return a JSON formatted string that represents the current BuildManif
        public string ToJson();
        // Return an INI formatted string that represents the current BuildManif
        public override string ToString();
    }
}
```

Cloud Build REST API

Leave feedback

You can further automate and integrate **Unity Cloud Build** into your own custom workflows using its REST API. The API can give you additional insight into your build history and allows you to create webhook callbacks to facilitate communication between __Unity Cloud Build__'s servers and your own.

See the Unity **Cloud Build** API documentation on the Unity Cloud Build website for further details.

Unity IAP

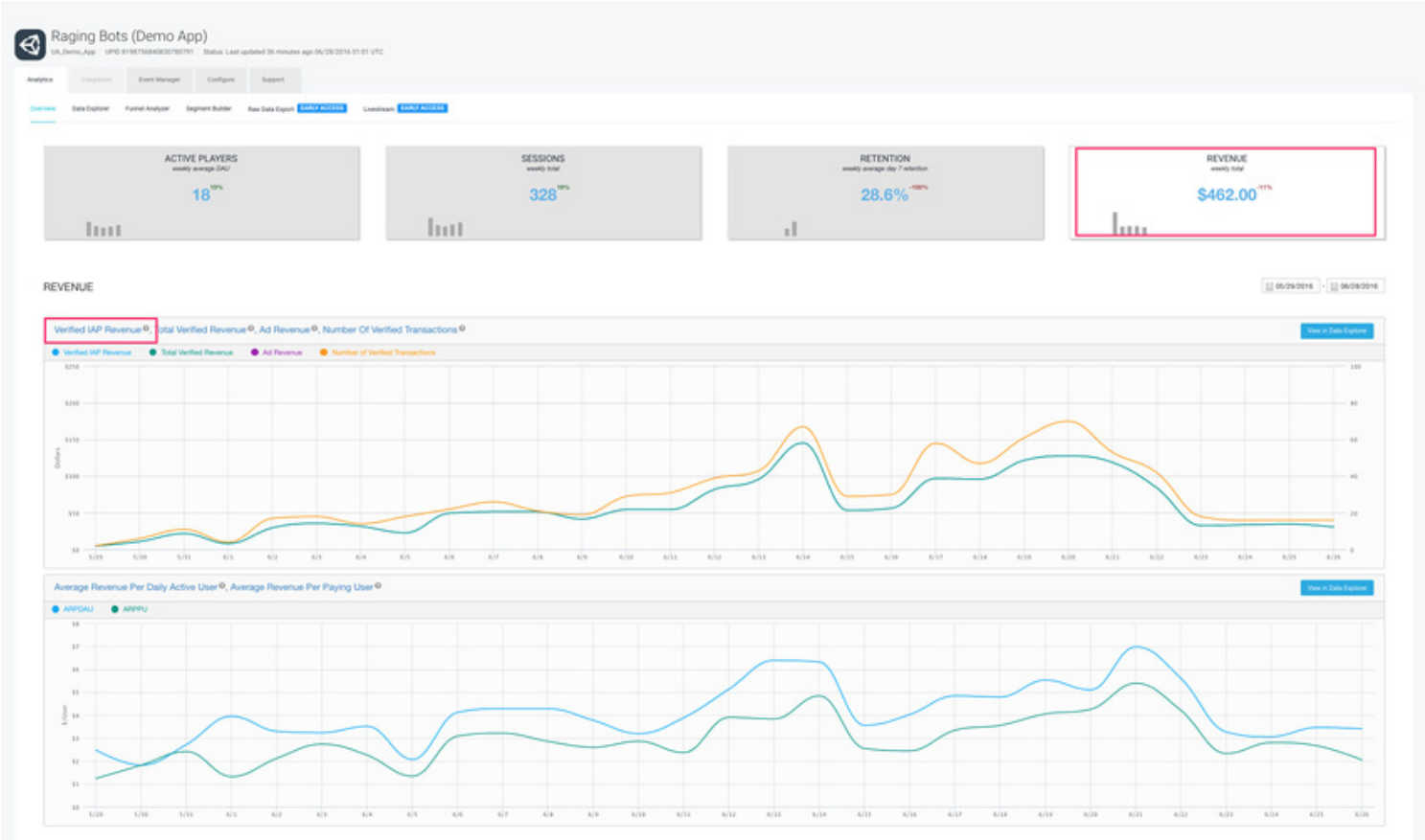
Leave feedback

Unity IAP makes it easy to implement in-app purchases in your application across the most popular App stores.

Supported platforms include:

iOS App Store
Mac App Store
Google Play
Universal Windows Platform
Amazon Appstore
Samsung Galaxy Apps
Tizen Store
CloudMoolah Store
Facebook Gameroom
Xiaomi Mi Game Pay

You can view IAP revenue through the Unity Analytics Dashboard pictured below.



Unity Analytics dashboard

2017–09–01 Page published with editorial review

Updated IAP supported platform list in Unity 2017.1

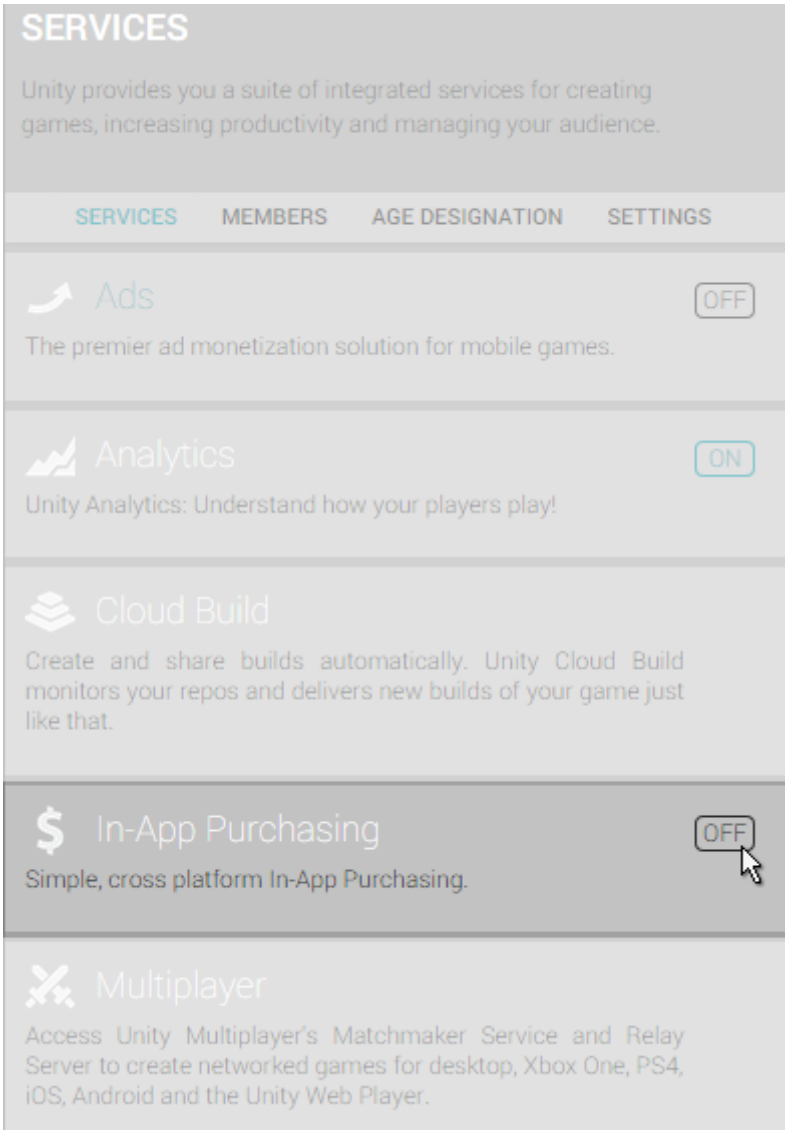
Setting up Unity IAP

Leave feedback

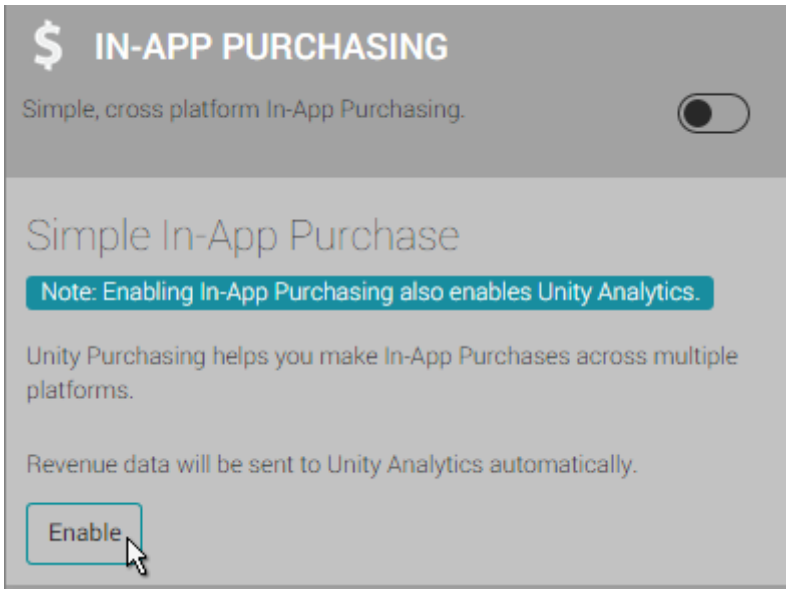
Once you have set up your project for Unity Services, you can enable the **Unity IAP** service.

Step 1. Enable In-App Purchasing

In the **Services window**, select **In-App Purchasing**.



Click the **Enable** button to enable In-App Purchasing.



Note: Common Unity IAP integration compiler errors

The following error messages may indicate that Unity IAP is disabled in the Unity Cloud Services window, or that Unity is disconnected from the Internet:

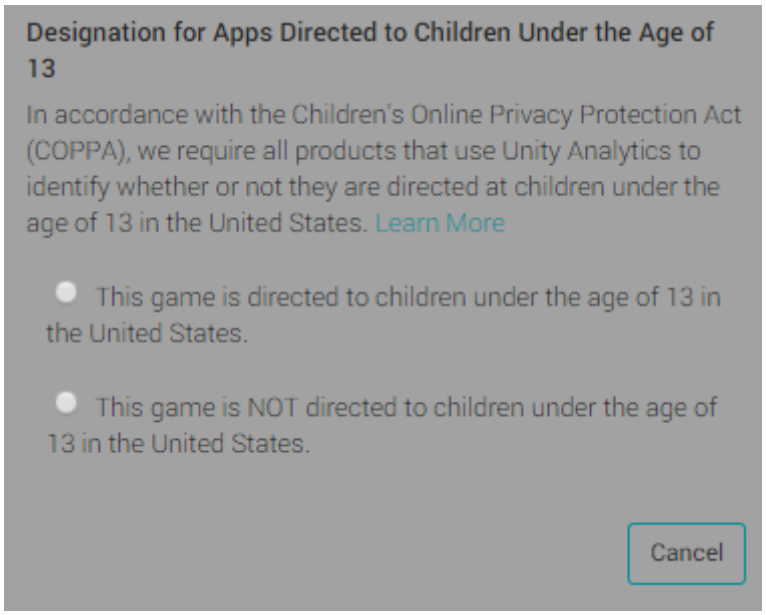
CS0246: The type or namespace name IPurchaseReceipt could not be found.
System.Reflection.ReflectionTypeLoadException
UnityPurchasing/Bin/Stores.dll
UnityEngine.Purchasing

To resolve these errors, first try reloading the Services window. A quick way to do this is to close it and then reopen it. Once reloaded, make sure that Unity IAP is enabled.

If this doesn't work, try disconnecting and reconnecting to the Internet, then sign back into Unity Services and re-enable Unity IAP. Only users with Unity Services "owner" or "manager" roles for the registered organisation can enable the Unity IAP Service.

Step 2. COPPA Compliance

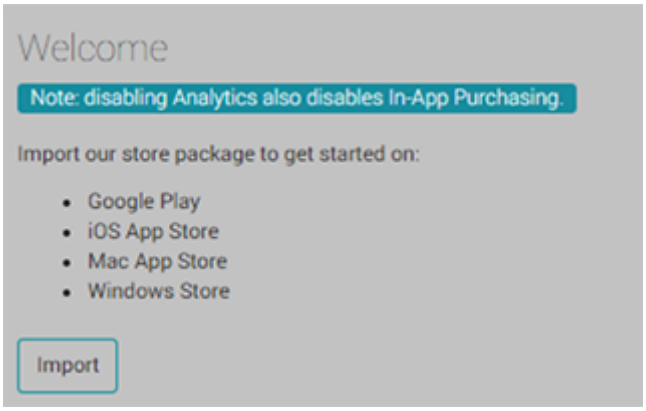
The Children's Online Privacy Protection Act (COPPA) applies to the online collection of personal information from children under 13. The rules spell out what you must include in a privacy policy, when and how to seek verifiable consent from a parent, and what responsibilities you have to protect children's privacy and safety online. If you have not already specified a **COPPA** choice in your **Analytics** settings, a dialog window will appear asking about the target age for users of your app in order to ensure **COPPA** compliance. Choose the appropriate answer and then click **Save Changes**.



The COPPA Compliance target age prompt

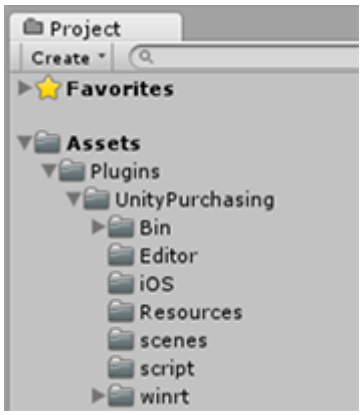
Step 3. Adding the IAP package

To import the Unity IAP package into your project, click **Import**.



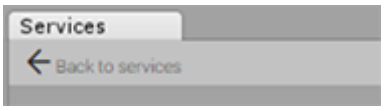
The IAP Import Package option

When you import the package, a new folder called **Plugins** is automatically added to your project. This folder contains UnityPurchasing **assets** required to use Unity IAP.



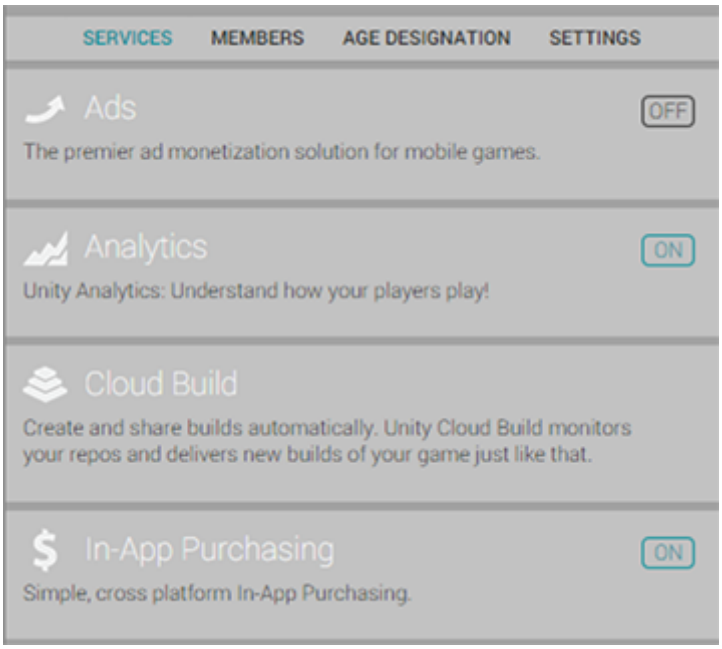
The imported IAP package files in the project window

Click **Back to services** to review the Services panel.



The Services window back button

Make sure that that Analytics and In-App Purchasing are both labelled **ON** as shown below.



Services window showing IAP and Analytics switched on

You can now begin implementing Unity In-App Purchases into your project.

Configuring for Apple App Store and Mac App Store

Leave feedback

Introduction

This guide describes the process of establishing the digital records and relationships necessary for a Unity game to interact with an In-App Purchase Store. The Unity IAP purchasing API is targeted.

In-App Purchase (IAP) is the process of transacting money for digital goods. A platform's Store allows purchase of Products, representing digital goods. These Products have an Identifier, typically of string datatype. Products have Types to represent their durability: *subscription*, *consumable* (capable of being rebought), and *non-consumable* (capable of being bought only once) are the most common.

Apple App Store

Getting Started

Write a game implementing Unity IAP. See Unity IAP Initialization and Integrating Unity IAP with your game.

Keep the game's product identifiers on-hand for use in iTunes Connect later.

```
public class MyStoreClass : MonoBehaviour, IStoreListener {
    void Start() {
        var module = StandardPurchasingModule.Instance();
        ConfigurationBuilder builder = ConfigurationBuilder.Instance(module);
        builder.AddProduct("com.unity3d.storeguidedemo.100gold", ProductType.Consumable);
        UnityPurchasing.Initialize(this, builder);
    }
```

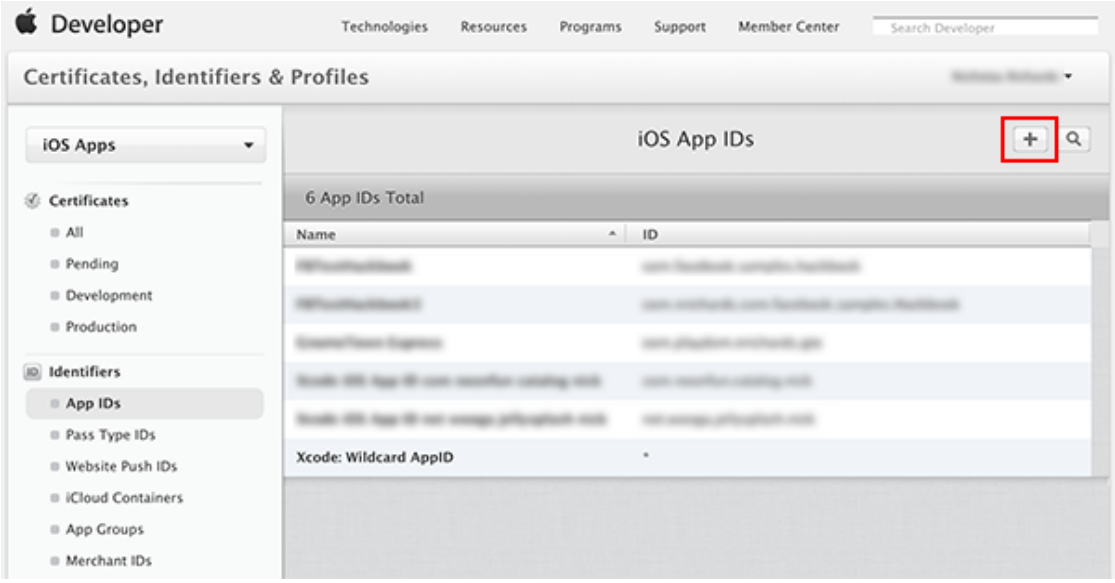
Register the Application

In the Apple Developer Center, navigate to the appropriate Identifiers section.

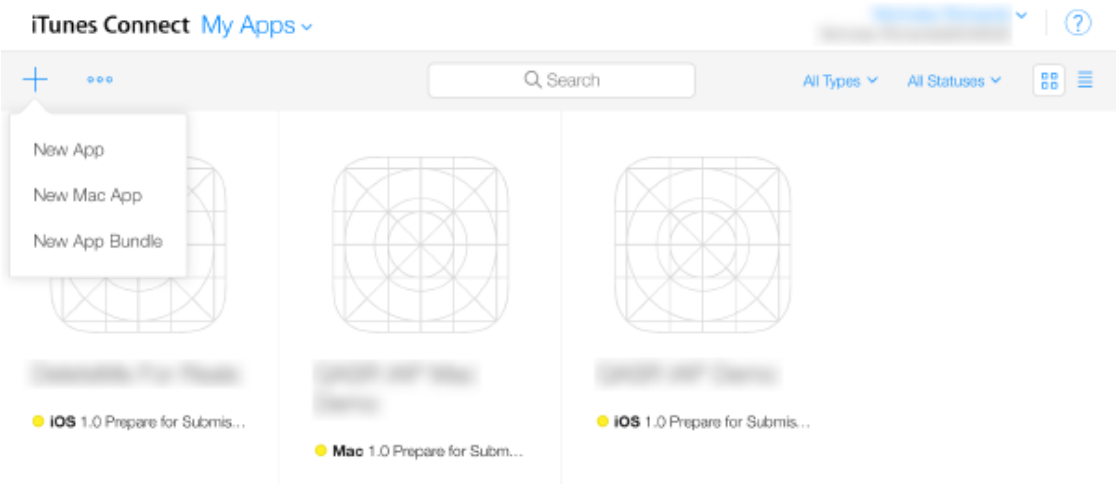
Add a new App ID to create a fundamental application entity with Apple.

NOTE: Use an Explicit App ID. Wildcard App IDs (com.example.*) cannot be used for applications that use In-App Purchases.

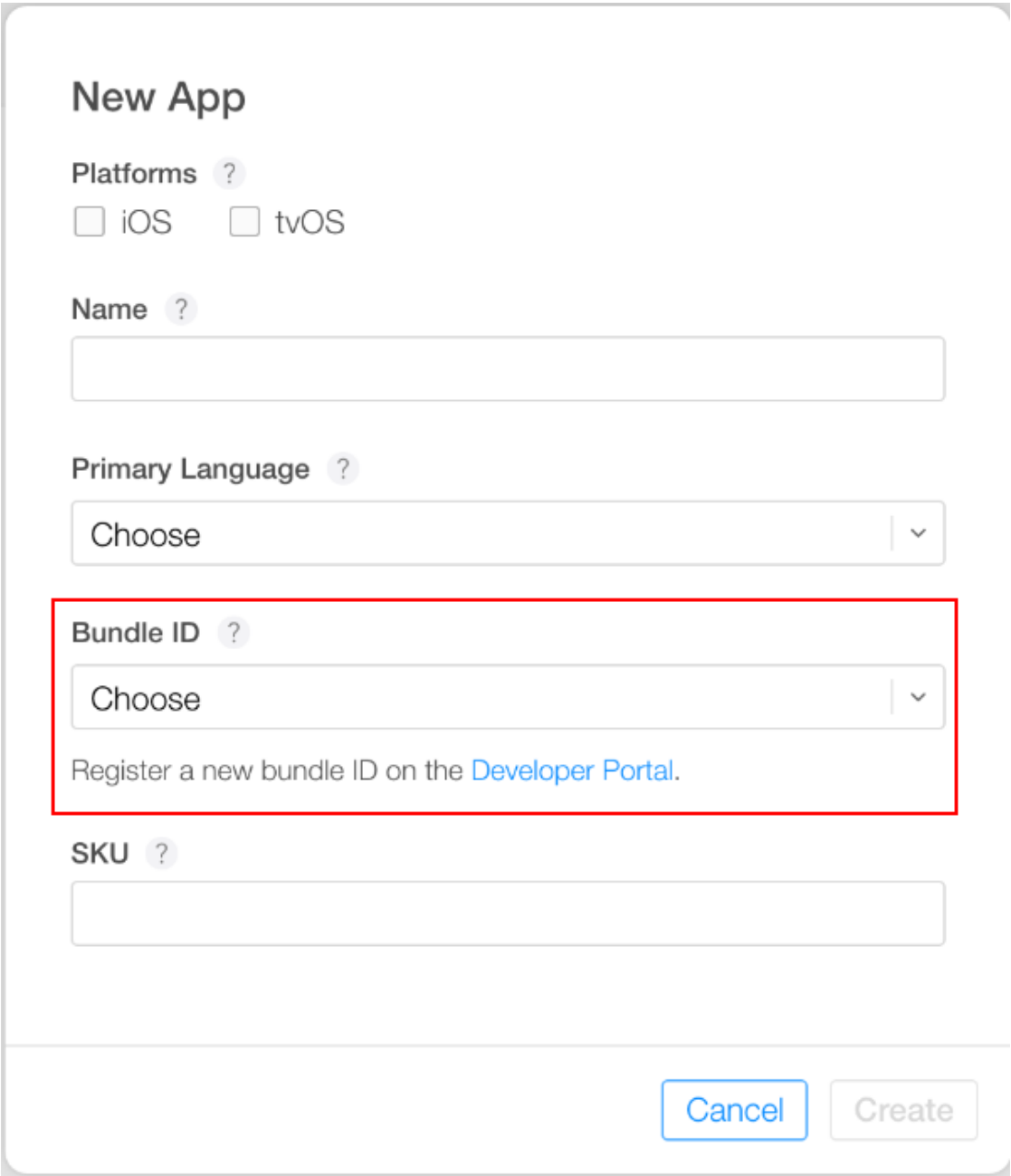
NOTE: The App ID is available to use in iTunes Connect after you create it in the Developer Center.



Navigate to iTunes Connect and create an App, to establish a Store relationship with your game.

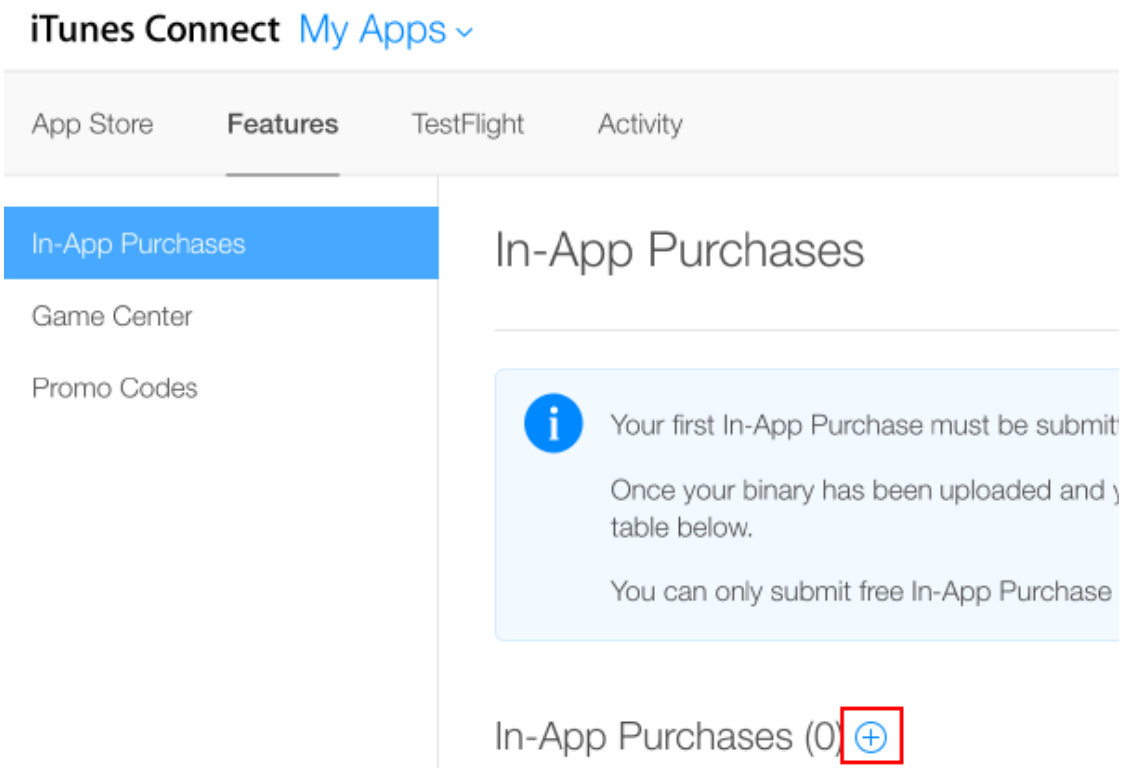


Use the newly created App ID for the app's Bundle ID.

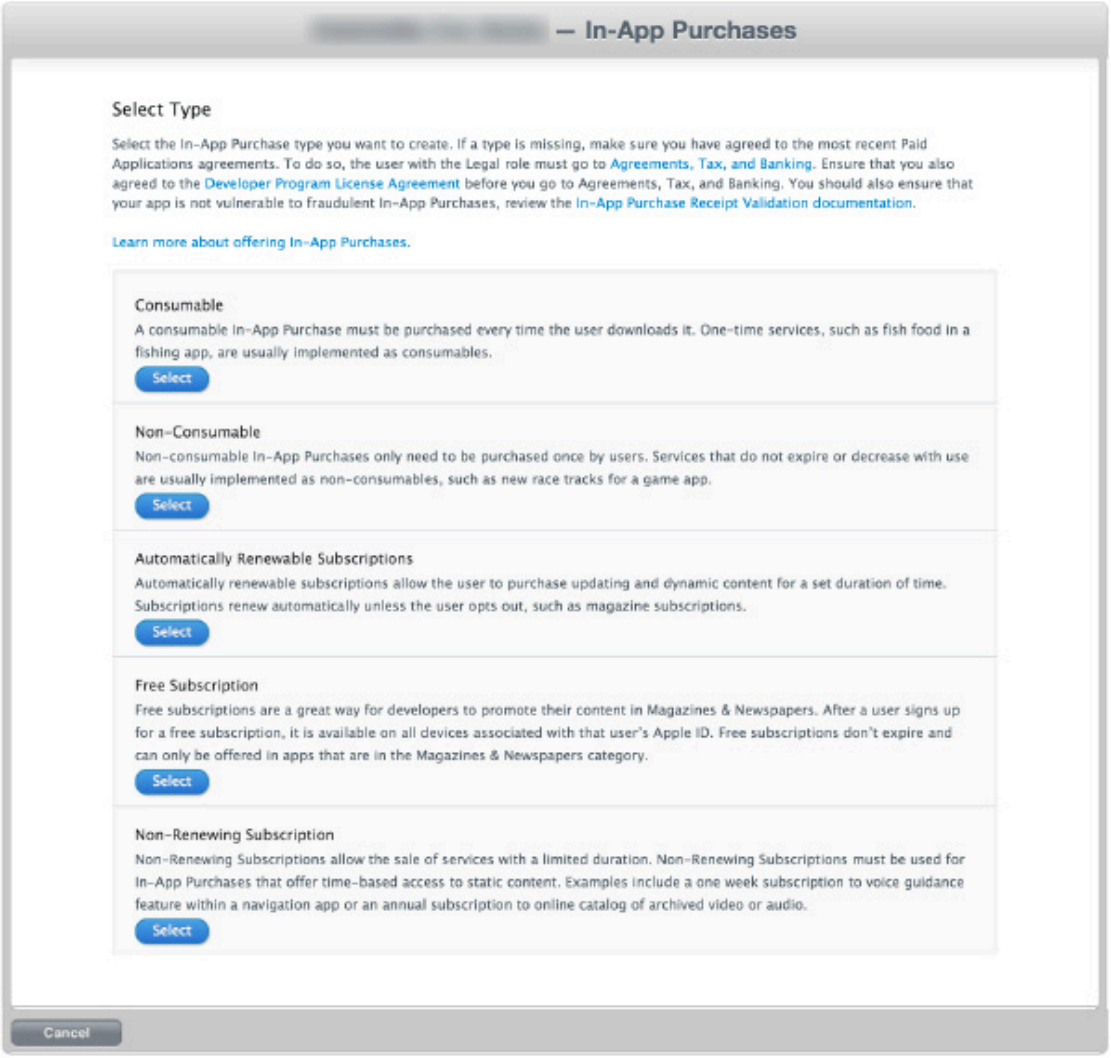


Add In-App Purchases

Choose **Features** and add a new In-App Purchase with the plus ("+") button.



Choose a Product Type.



Specify the Product Identifier, and complete other fields as requested.

NOTE: The "Product ID" here is the same identifier used in the game source code, added to the Unity IAP ConfigurationBuilder instance via **AddProduct()** or **AddProducts()**.

NOTE: When targeting multiple Apple device groups (for example, shipping on both iOS and Mac) Apple requires usage of different, unique product identifiers for each distinct device group. Use Unity IAP's Purchasing.IDs class and define a one-to-many mapping Product IDs to the store-specific identifiers, and pass that mapping in when initializing IAP.

— In-App Purchases

In-App Purchase Summary

Enter a reference name and a product ID for this In-App Purchase.

Reference Name:

Product ID:

Pricing and Availability

Enter the pricing and availability details for this In-App Purchase below.

Cleared for Sale **Yes** **No**

Price Tier Select

View Pricing Matrix

In-App Purchase Details

Language

Details for this In-App Purchase are shown below. You must provide at least one language at all times.

Add Language

Language	Display Name	Description
Click Add Language to get started.		

Review Notes (Optional)

Additional information about your In-App Purchase that can help us with our review, such as test accounts that can be used (including user names, passwords and so on). Review notes cannot exceed 4000 bytes.

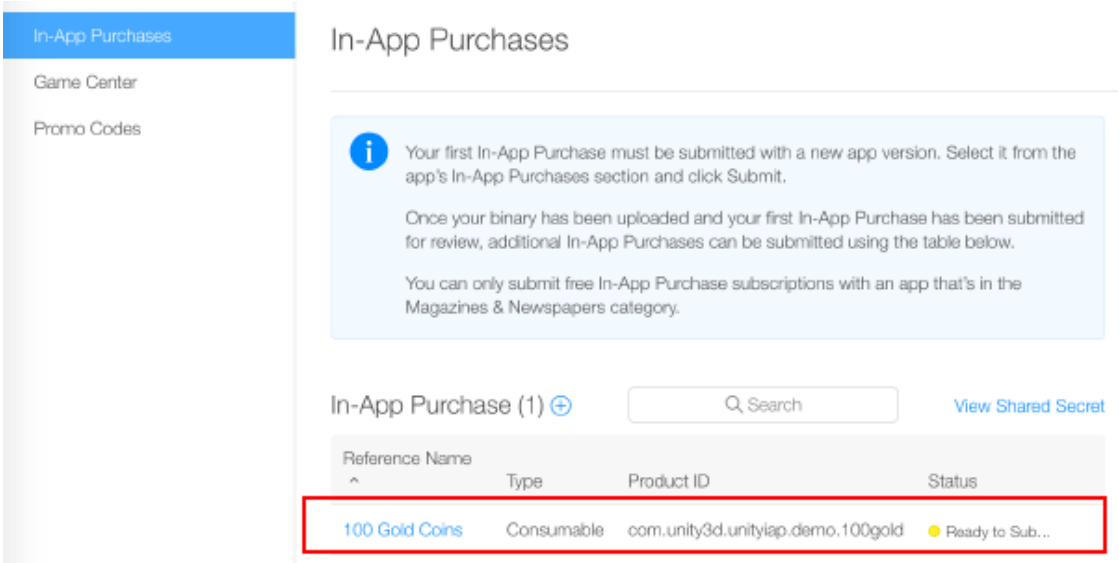
Screenshot for Review

Before you submit your In-App Purchase for review, you must upload a screenshot. This screenshot will be for review purposes only. It will not be displayed on the App Store. Screenshots must be at least 312x390 pixels and at least 72 DPI.

Choose File

Cancel Save

Result:

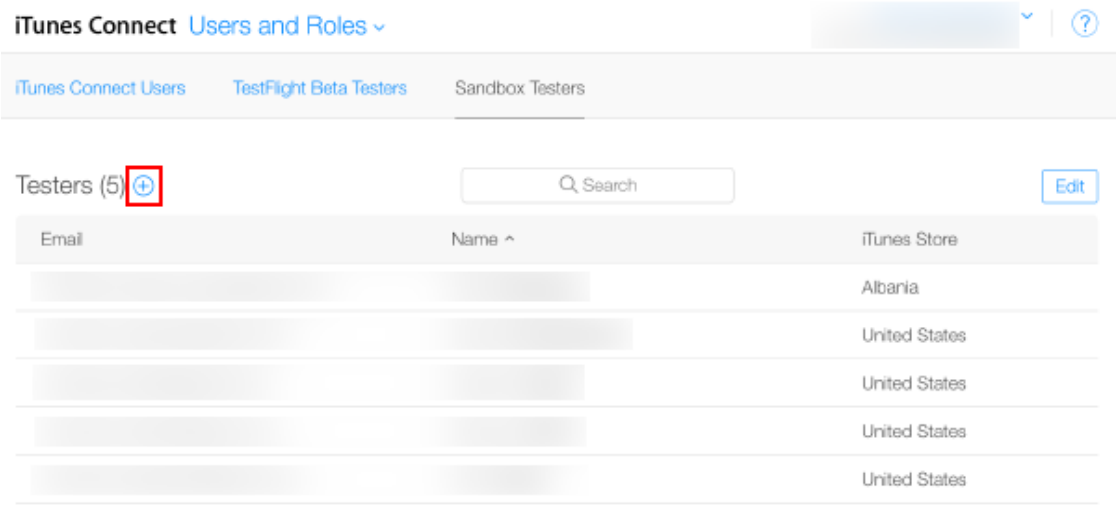


Test IAP

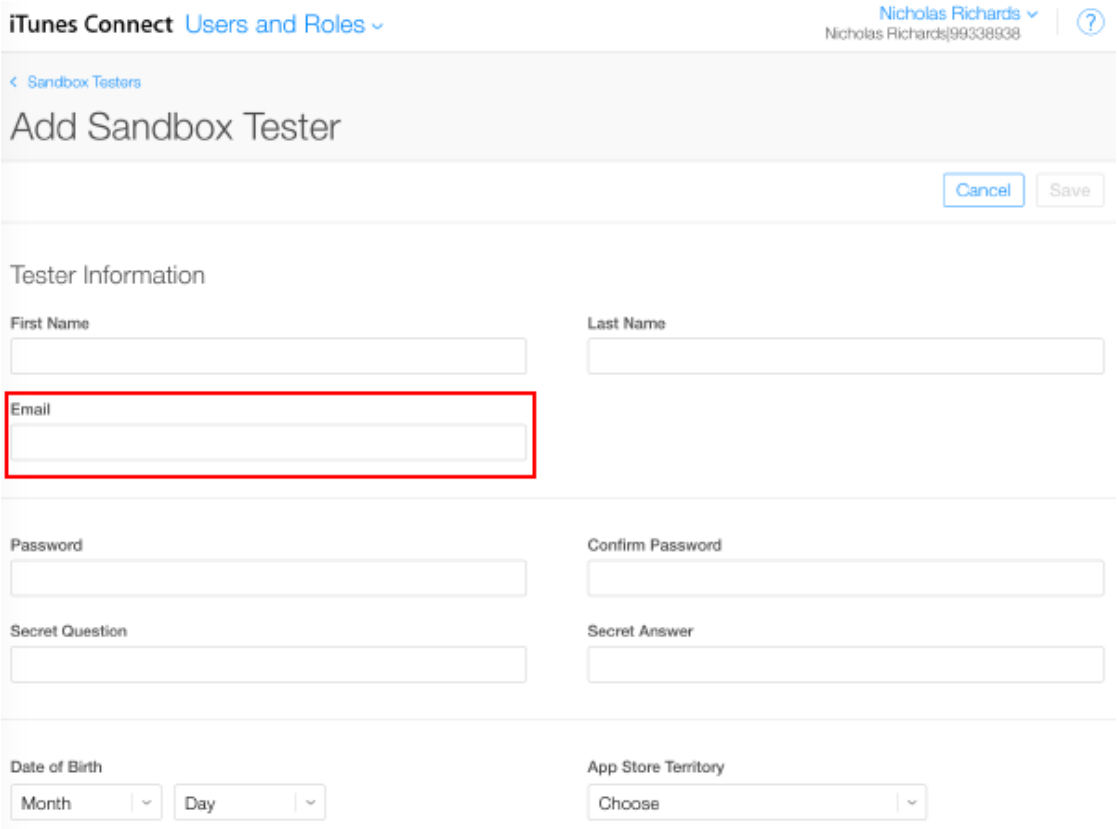
Create **Sandbox Testers** using iTunes Connect for use on your test device's iTunes Account. To do this, navigate to **iTunes Connect > Users and Roles**, and choose the plus ("+") button. You must review Apple's Sandbox Tester documentation as there are several additional important usage notes, and you must use a real email address to create Testers.

NOTE: See the iOS and Mac App Store guides for additional details.

TIP: (*) To simplify managing the email address, use an email service capable of sub-addressing (emailaccount+subaddress@example.com) such as Gmail, iCloud, and Outlook.com. This allows one email account to receive email for multiple sub-addresses.



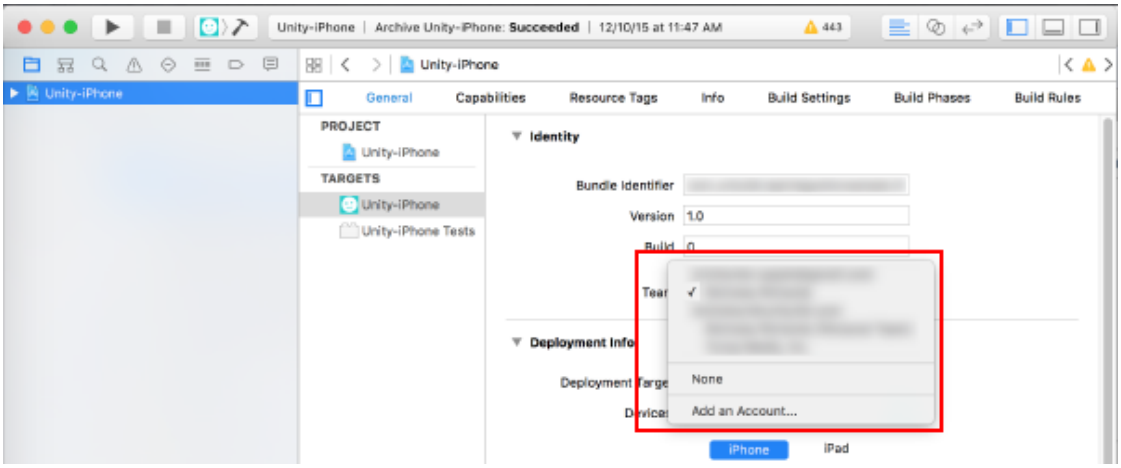
Walk through the user creation wizard.



Build the Xcode project for your game using Unity.

NOTE: Ensure the Bundle Identifier in Unity (Build Settings > iOS > Settings ... > Other Settings > Bundle Identifier) matches that used in iTunes Connect.

Then, in your game's Xcode project, ensure the Team (Project Navigator > your game Target > General > Identity > Team) is set to that of your Apple Developer account.



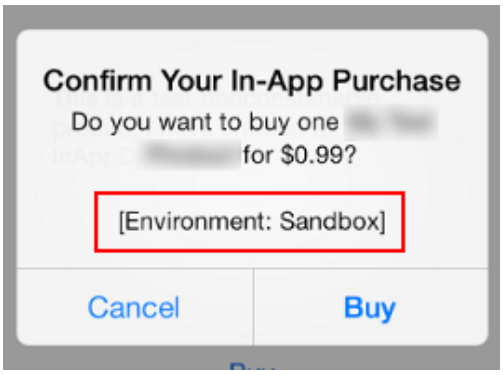
For iOS

Using the target **iOS** device, sign out of any existing Apple ID accounts. Only sign in as the Sandbox Tester when prompted by the app, later. Any subsequent purchases are routed through the Apple Sandbox instead of the Production Store.

Build and run the game on your iOS device. `UnityPurchasing.Initialize()` succeeds if everything has been correctly configured. See Unity Purchasing Initialization

Test the IAP by making a purchase in the game on the device. A modified purchase dialog displays, explaining that this purchase is being performed in the Sandbox Environment. Use the Sandbox User Tester password when prompted for purchase.

WARNING: If the indicator is not present, then an account is charged real money for the product.



For Mac

When building a desktop Mac build, select **Mac App Store Validation** within Unity's Mac **Player Settings**.

Once you have built your App, update its `info.plist` file with your bundle identifier and version strings. Right click on the .app file and click **Show Package Contents**, locate the `info.plist` file and update the `CFBundleIdentifier` string to your application's bundle identifier.

Sign, package, and install your application. Run the following commands from an OSX terminal, filling in “your.app” and “your.pkg” appropriately.

TIP: To sign the bundle, you may first need to remove the Contents.meta file if it exists:
`your.app/Contents/Plugins/unitypurchasing.bundle/Contents.meta`

```
codesign -f --deep -s "3rd Party Mac Developer Application: "
your.app/Contents/Plugins/unitypurchasing.bundle

codesign -f --deep -s "3rd Party Mac Developer Application: " your.app

productbuild --component your.app /Applications --sign "3rd Party Mac Developer Installer: "
your.pkg
```

To install the package correctly, delete the unpackaged .app file before running the newly created package and installing it.

Launch the app from the *Applications* folder. The first time you do so, you are prompted to enter your iTunes account details, for which you can then make test purchases against the sandbox environment.

See pages on iOS and Mac Extended Functionality and Delivering applications to the Apple Mac Store for additional details on Apple App Store testing and signing.

Configuring for Google Play Store

Leave feedback

Introduction

This guide describes the process of establishing the digital records and relationships necessary for a Unity game to interact with an In-App Purchase Store. The Unity IAP purchasing API is targeted.

In-App Purchase (IAP) is the process of transacting money for digital goods. A platform's Store allows the purchase of Products, representing digital goods. These Products have an Identifier, typically of string datatype. Products have Types to represent their durability: *subscription*, *consumable* (capable of being rebought), and *non-consumable* (capable of being bought once) are the most common.

Google Play Store

Getting Started

Write a game implementing Unity IAP. See Unity IAP Initialization and Integrating Unity IAP with your game.

Keep the game's product identifiers on-hand for Google Play Developer Console use later.

```
public class MyStoreClass : MonoBehaviour, IStoreListener {
    void Start() {
        var module = StandardPurchasingModule.Instance();
        ConfigurationBuilder builder = ConfigurationBuilder.Instance(module);
        builder.AddProduct("com.unity3d.storeguidedemo.100gold", ProductType.Consumable);
        UnityPurchasing.Initialize(this, builder);
    }
```

Build a signed non-Development Build Android APK from your game.

TIP: Make sure you safely store your **keystore** file. The original **keystore** is always required to update a published Google Play application.

TIP: Reuse the Bundle Version Code from your last uploaded **APK** during local testing to permit side-loading without first being required to upload the changed **APK** to the Developer Console. See Android Player Settings.

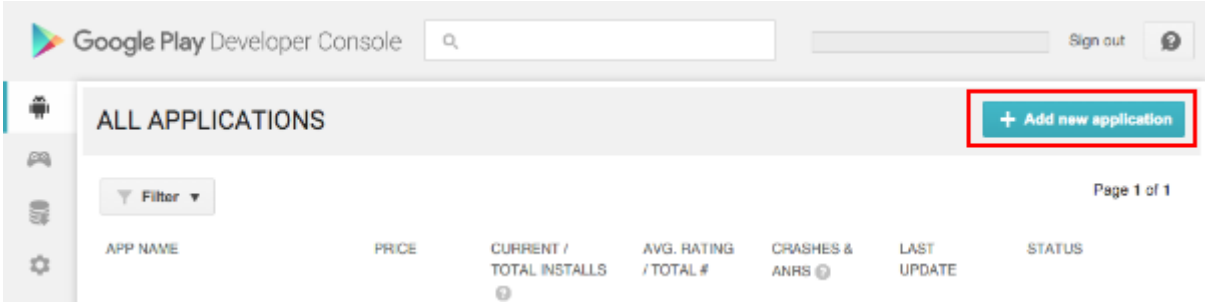
Register the Application

From the Google Account that will publish the game, register the Android application with the Google Play Developer Console.

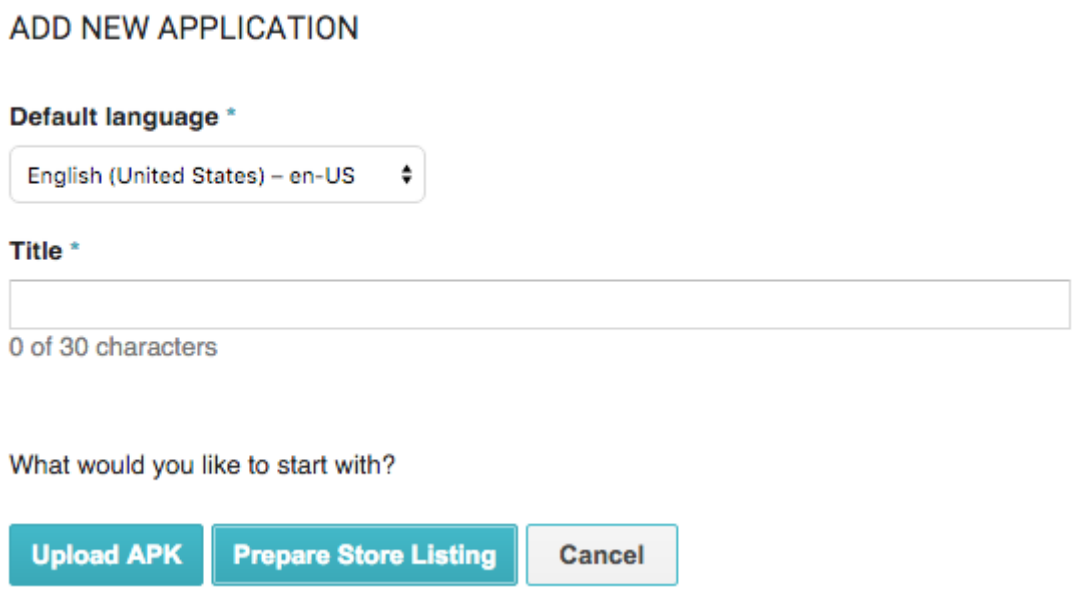
NOTE: This guide uses the Google Play Alpha/Beta Testing approach for testing in-app purchase integration.

NOTE: A restriction against using the publisher's Google Account for testing exists, because Google payments does not let you buy items from yourself. Therefore you will need to create a non-publisher Google Account for testing.

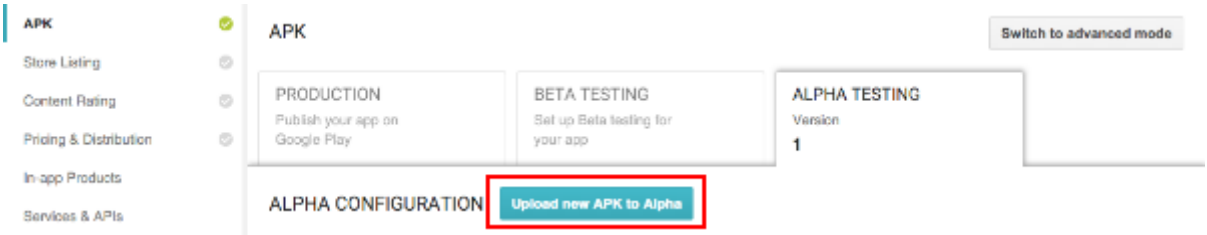
Choose **Add new application**.



Give the application a Title. For configuring application billing, choose “Upload APK” now. The Store Listing will be prepared and populated in a later step.



Navigate to **ALPHA TESTING** and choose **Upload your first APK to Alpha**. Select your APK and upload it.



Resolve any issues listed in the **Why can’t I publish?** popup first, then click **Publish app** to publish your Alpha application.

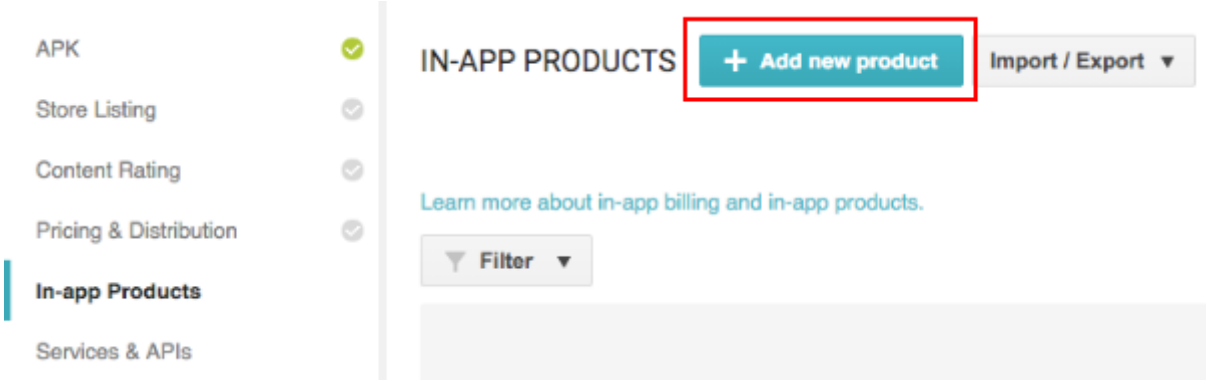
TIP: There may be a delay of up to 24 hours while the published Alpha application is made available for download from the Google Play Store. Similarly, there may be a 24-hour delay to published changes to an application’s In-App Purchases (see below).

WARNING: Make sure the published application is the Alpha version. If the Production version is published, the general public will have access to the untested application.

Add In-App Purchases

In the Developer Console, add one or more in-app purchases for the game.

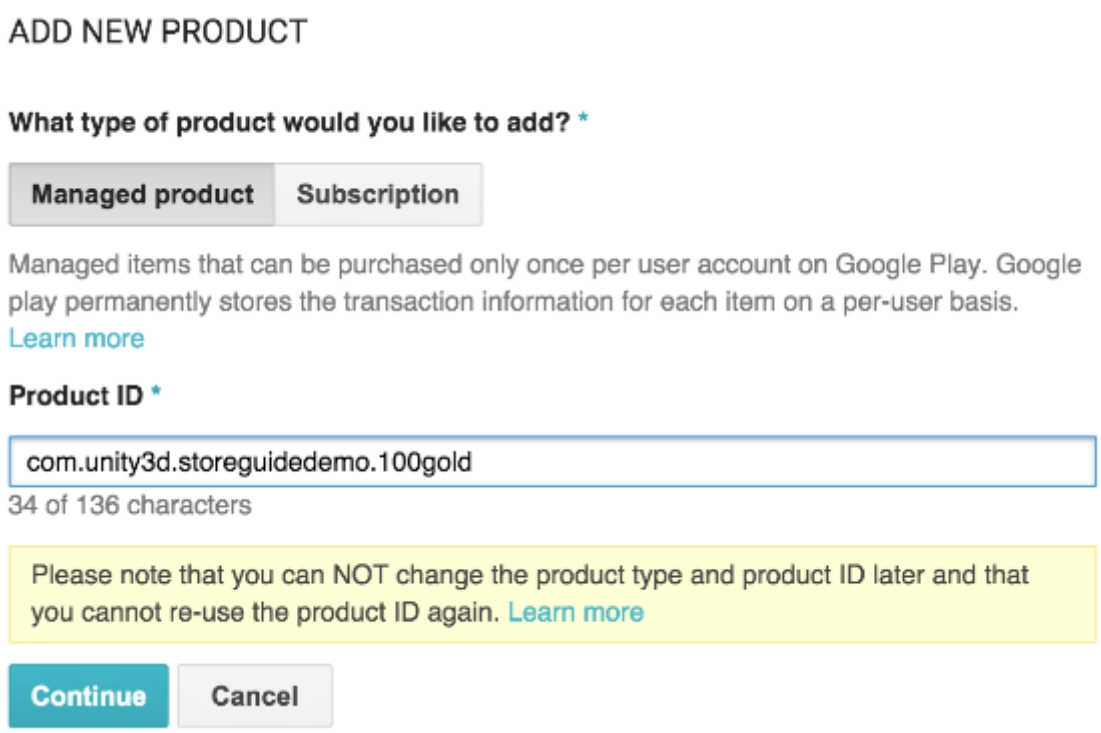
Navigate to **In-app Products** and choose **Add new product**.



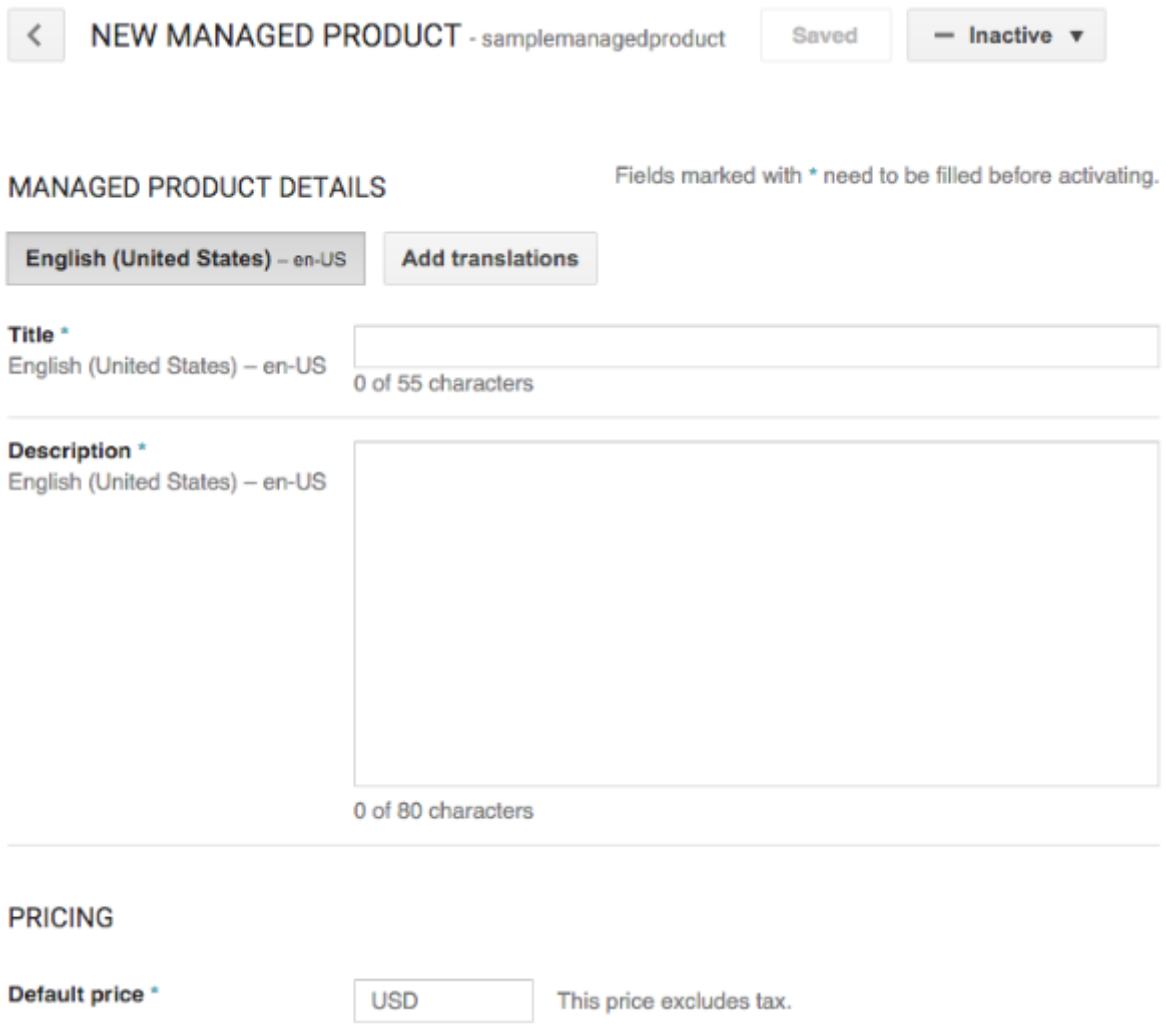
Define the **Product ID** and choose **Continue**.

You can specify a consumable or non-consumable Product Type in **Managed product**. **Subscription** is also supported by Unity IAP.

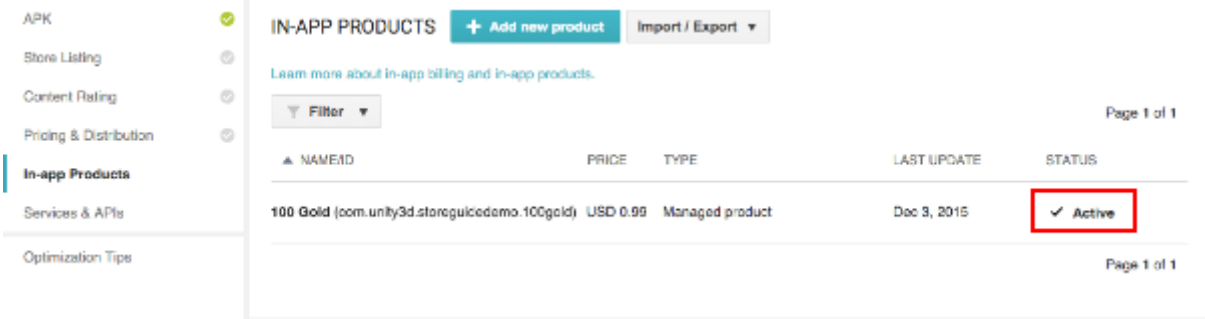
NOTE: The "Product ID" here is the same identifier used in the game source code, added to the Unity IAP ConfigurationBuilder instance via `AddProduct()` or `AddProducts()`.



Populate the IAP's **Title**, **Description**, and **Default price**. Then enable it by changing the **Inactive** button to **Active**.



Result:



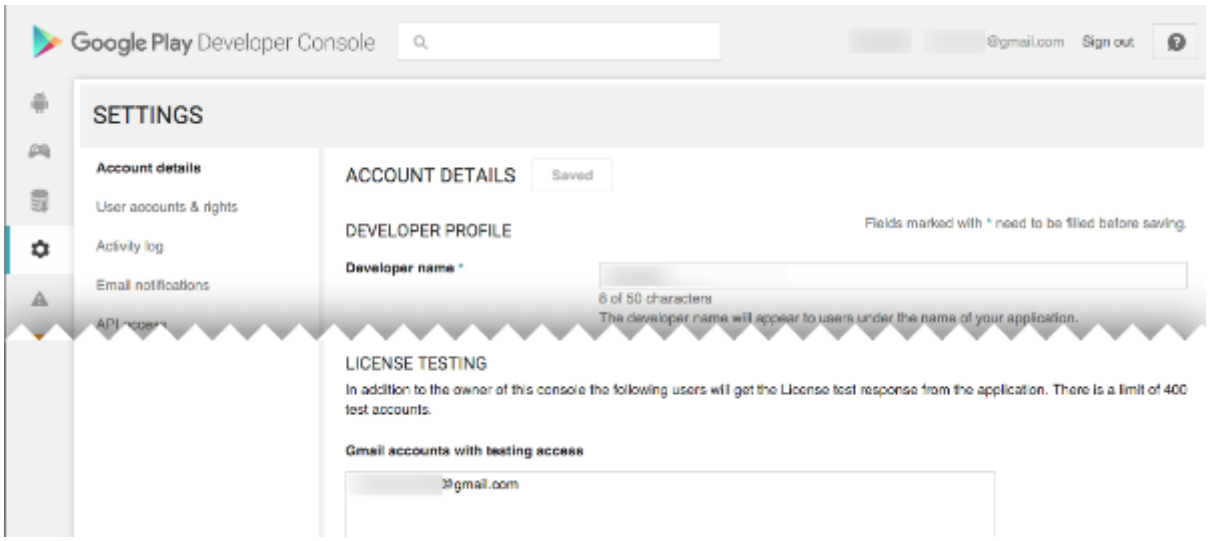
Test IAP

Follow these instructions to assign "test users" to be granted special license testing permissions, allowing them to purchase any IAP for free for this application, and register them as Alpha Testers.

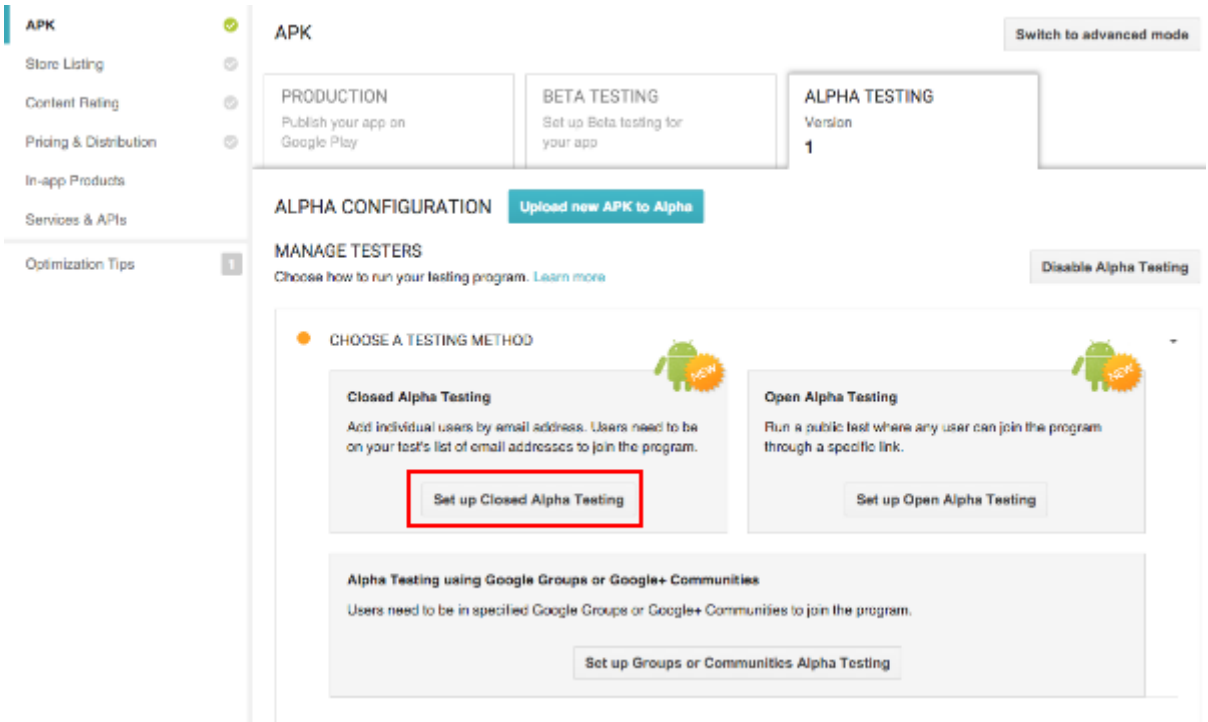
NOTE: Here we used Closed Alpha Testing. Other tester groupings are available.

Add Google Account email addresses for testers who will be granted a license to purchase any IAP for free via the Developer Console's **Settings > Account details > LICENSE TESTING** section.

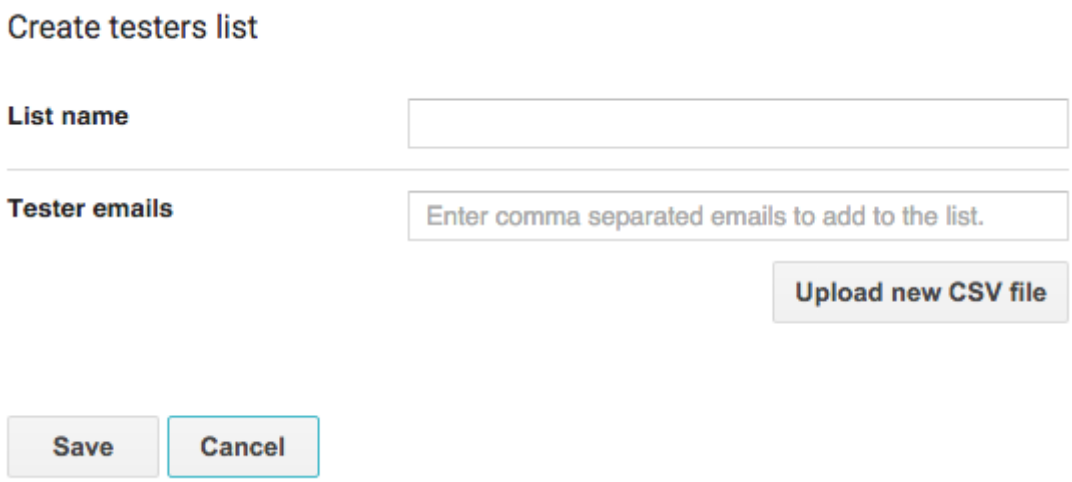
NOTE: There may be a delay of 15 minutes for the license testing setting to become effective.



Establish a list of Alpha Tester Google Account email addresses, from your list of license testing users.

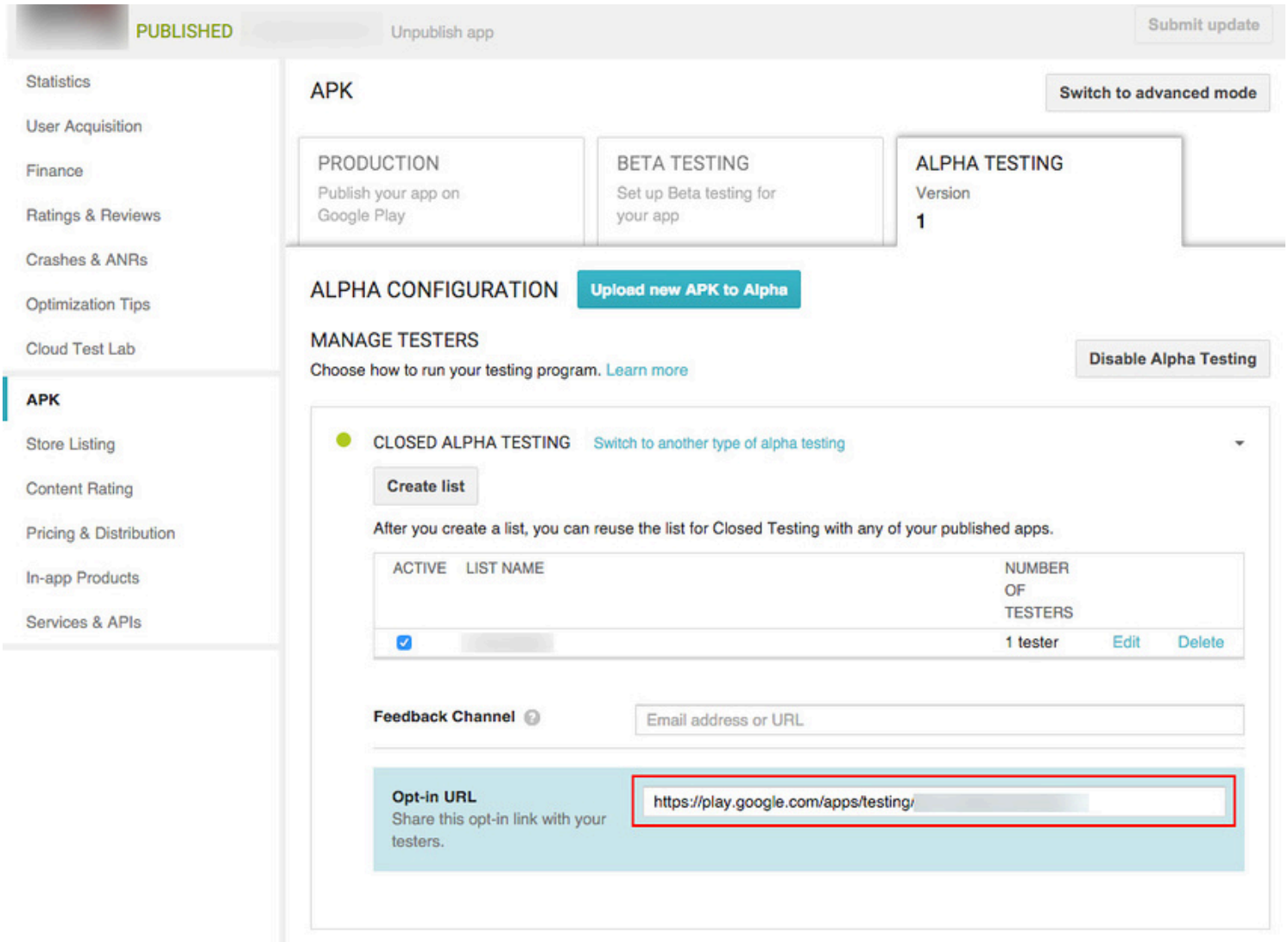


Add each Google Account email address and name the list.



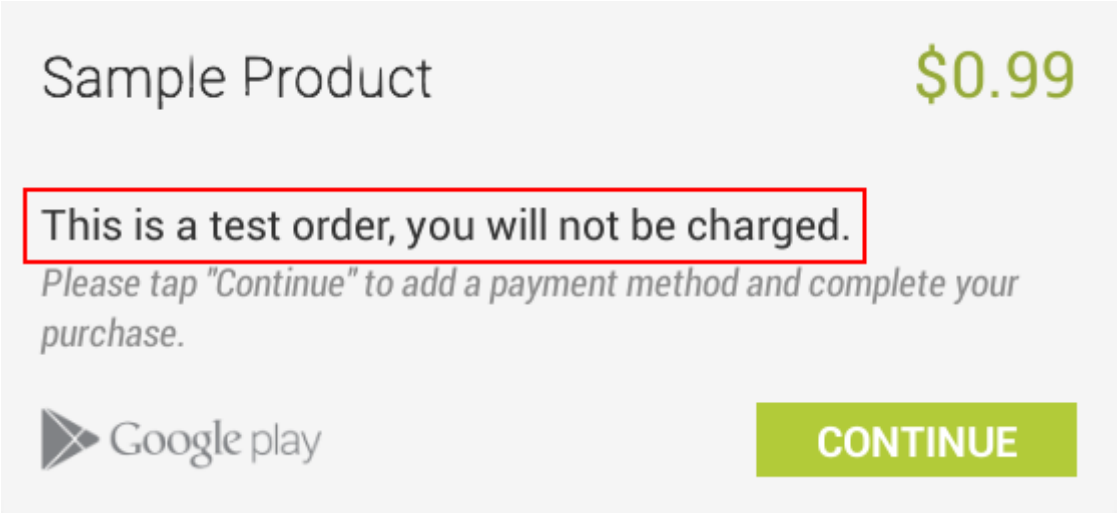
When available (after a multi-hour delay starting from publishing the APK) share the **Opt-in URL** with the Alpha Testers and have them install the application from the store.

NOTE: To test updates retaining permission to purchase IAP's for free, one may side-load applications, updating the existing store-downloaded APK install.



To test the IAP, make a purchase on a device logged in with an Alpha Tester Google Account. A modified purchase dialog box appears to confirm the fact this product is under test and is free.

WARNING: If this dialog box does not appear, then the Alpha Tester Google Account will be charged real money for the product.


Sample Product
$0.99
This is a test order, you will not be charged.
Please tap "Continue" to add a payment method and complete your purchase.
Google play
CONTINUE

Configuring for Windows Store

Leave feedback

Introduction

This guide describes the process of establishing the digital records and relationships necessary for a Unity game to interact with an In-App Purchase Store. The Unity IAP purchasing API is targeted.

In-App Purchase (IAP) is the process of transacting money for digital goods. A platform's Store allows purchase of Products representing digital goods. These Products have an Identifier, typically of string datatype. Products have Types to represent their durability: *subscription*, *consumable* (capable of being rebought), and *non-consumable* (capable of being bought once) are the most common.

Windows Store

Introduction

Windows App Development offers both local and remote Windows Store client-server IAP testing.

This page covers local testing with the emulator and a simulated billing system, then Windows Store testing which limits app publication visibility to those with the app's link.

NOTE: This guide targets Windows 10 Universal SDK. Other Windows targets are available.

NOTE: Unity versions 5.6+ support **IL2CPP** Windows builds. Using Unity IAP for **IL2CPP** on Windows with earlier versions of Unity generates a compilation error.

Getting started

Write a game implementing Unity IAP. See Unity IAP Initialization and Integrating Unity IAP with your game.

Keep the game's product identifiers on-hand for use in Microsoft's Windows Dev Center Dashboard to perform remote Windows Store testing later.

```
public class MyStoreClass : MonoBehaviour, IStoreListener {
    void Start() {
        var module = StandardPurchasingModule.Instance();
        ConfigurationBuilder builder = ConfigurationBuilder.Instance(module);
        builder.AddProduct("com.unity3d.storeguidedemo.100gold", ProductType.Consumable);
        UnityPurchasing.Initialize(this, builder);
    }
```

Test IAP locally

Microsoft offers a simulated billing system, permitting local testing of IAP. This removes the need to configure anything on the Windows Dev Center or communicate with the the Windows Store via the app for initial integration testing.

Configuring local testing is far simpler than for remote Store testing, although it requires temporary code changes to the app which need to be removed before app publication.

To test IAP locally:

Enable the simulated billing system in code where Unity IAP is initialized with its ConfigurationBuilder instance.

WARNING: Remove these code changes after testing, before publishing to the Store; otherwise the app will not transact any real money via IAP!

```
var builder = ConfigurationBuilder.Instance(StandardPurchasingModule.Instance());
builder.Configure<IMicrosoftConfiguration>().useMockBillingSystem = true;
```

Build the application in Unity for **Universal Windows Platform**.

Open the application in Visual Studio and run the Local Machine target for x86.

Test IAP.

Remove the simulated billing system from code.

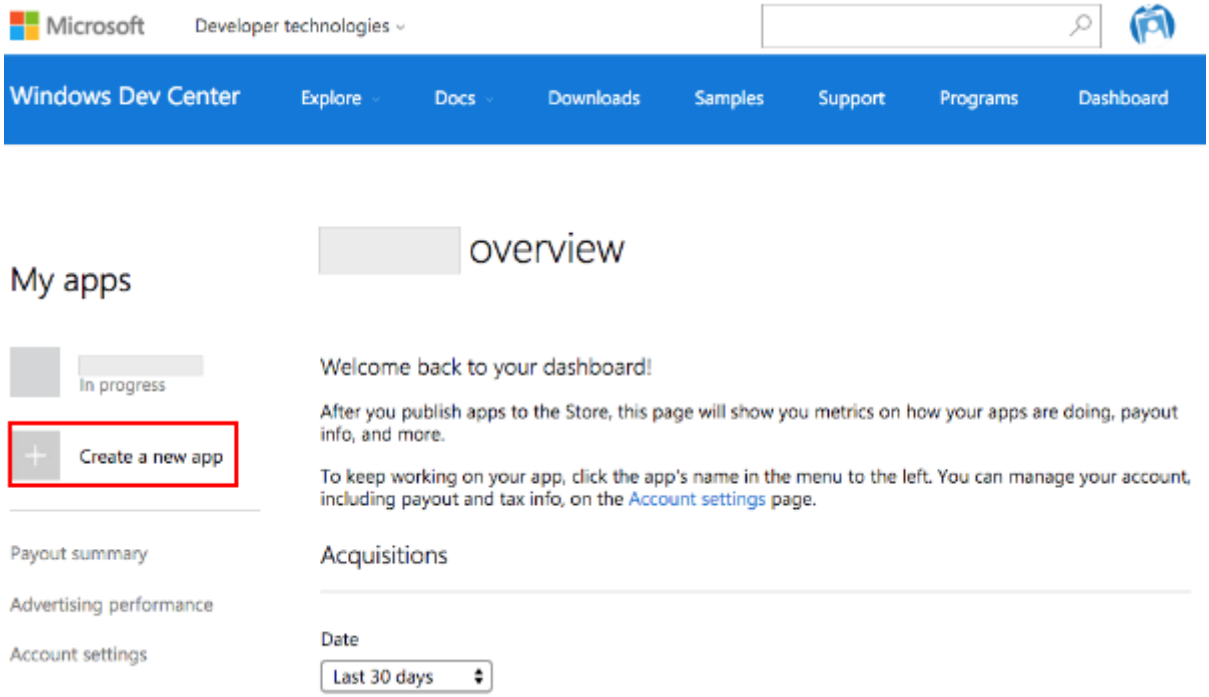
Register the App on the Windows Store

Once basic IAP functionality has been tested locally, you can more confidently begin working with the Windows Store. This test confirms that the app has all necessary IAPs registered correctly to permit purchasing.

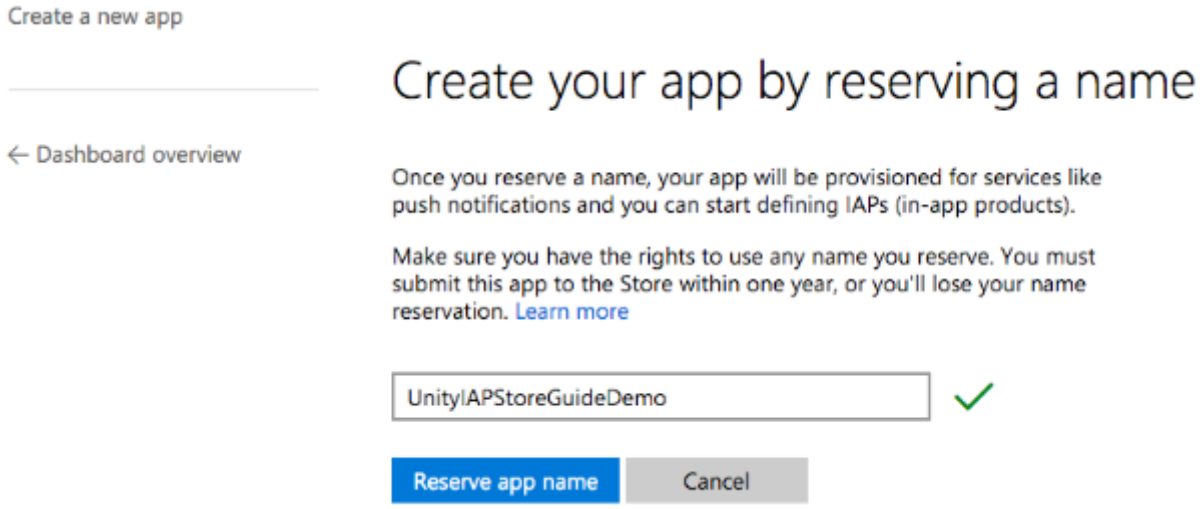
For testing IAP and publication use the Windows Dev Center and configure the app with a limited visibility. This limits the app's visibility to those who have its direct link.

NOTE: Testing on the Store also requires Certification, which may serve as an obstacle to testing. It is therefore important to complete testing locally before proceeding to testing with Windows Store.

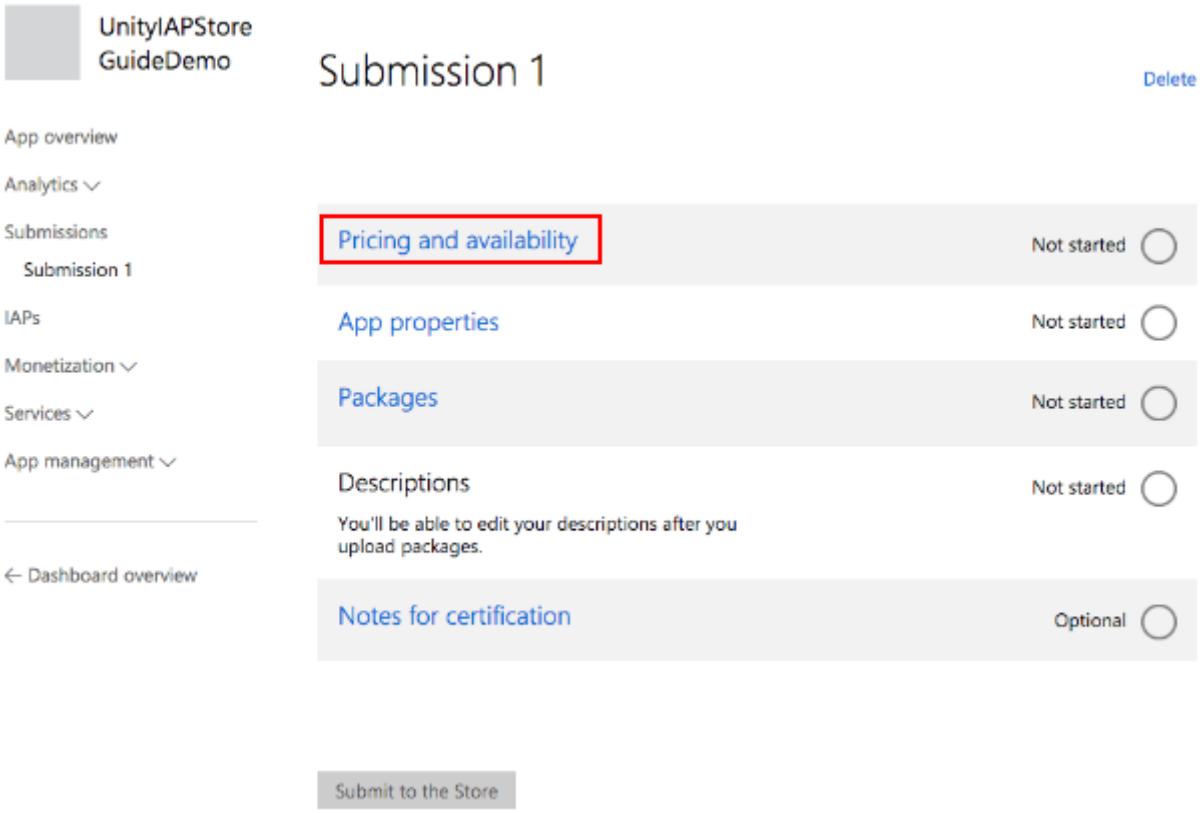
In the Dev Center create a new app.



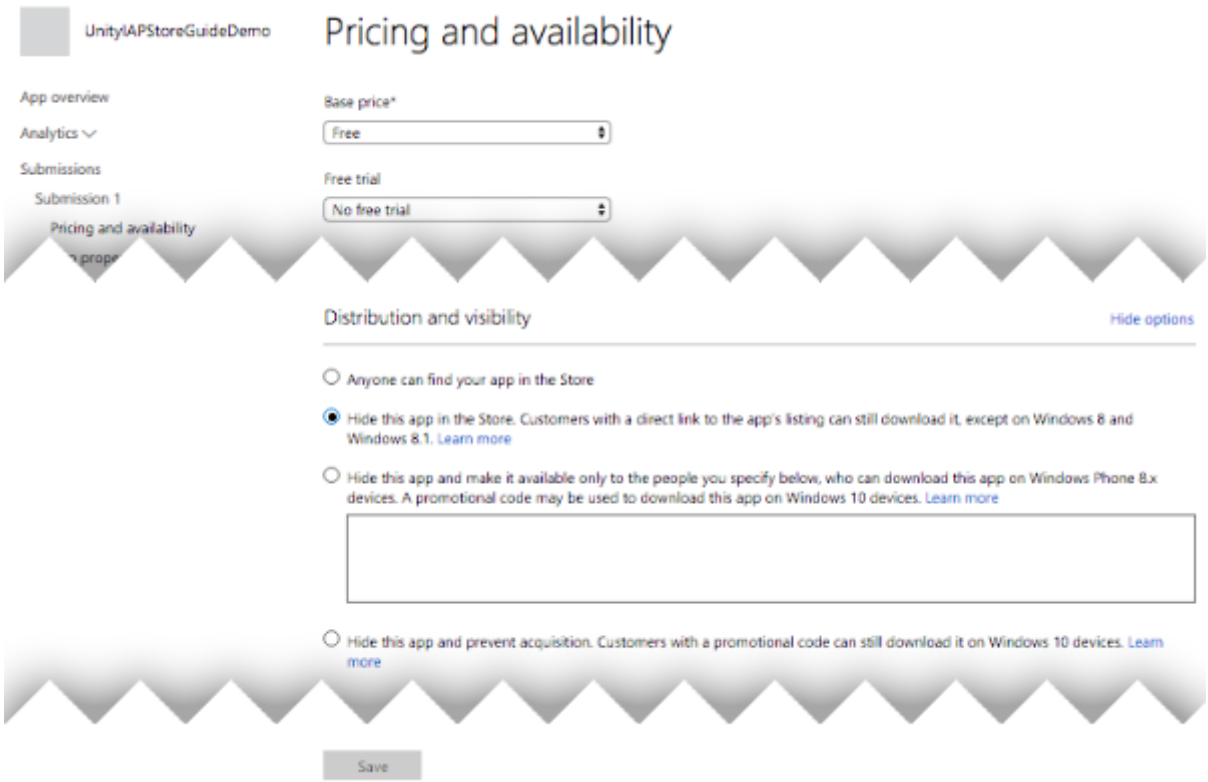
Reserve the app name.



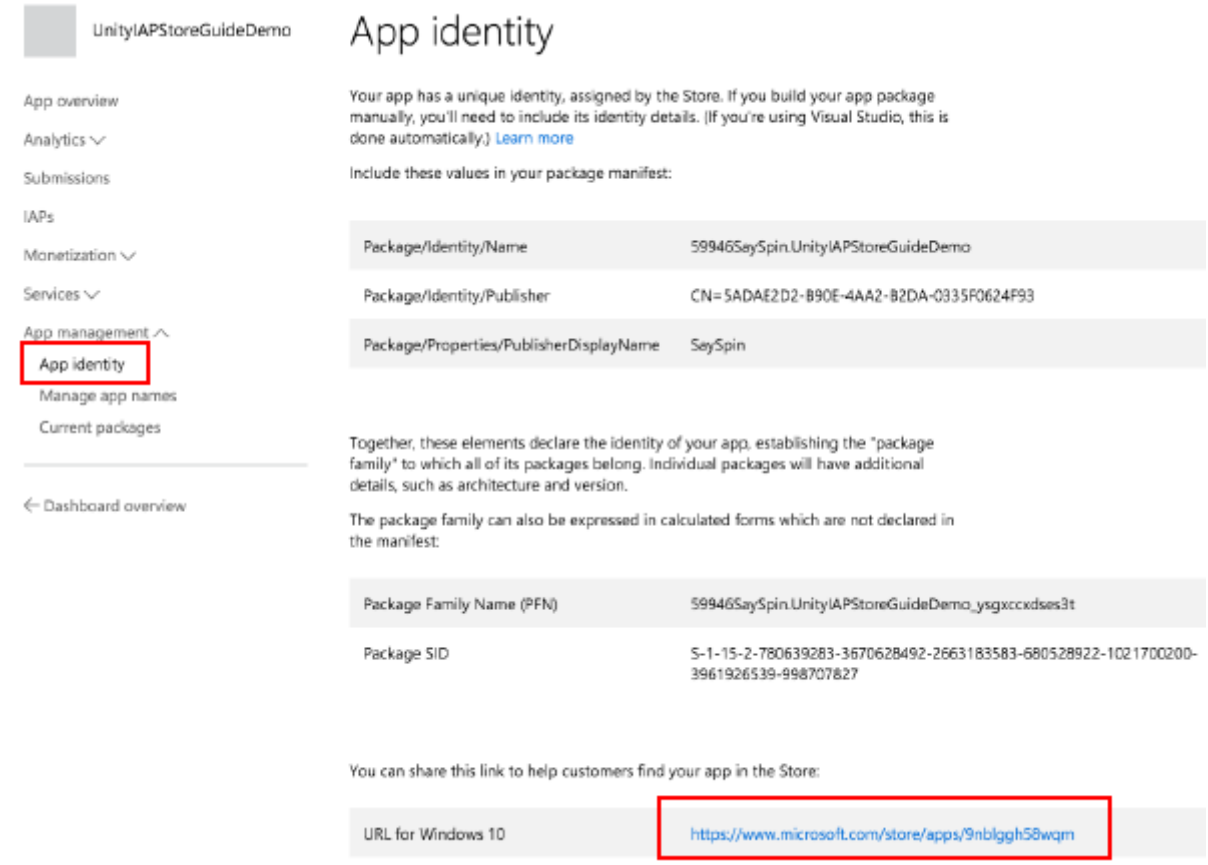
To test IAP with the Windows Store, the Windows Dev Center needs the published app. Click **Pricing and availability** and limit the app's Store visibility so that it is only available to users who have the app's direct link.



In "Distribution and visibility" see a list of the Store's available publication behaviors. Select **Hide this app in the Store**.



Collect the direct link. This will be used to install the app on a Windows 10 device for testing.



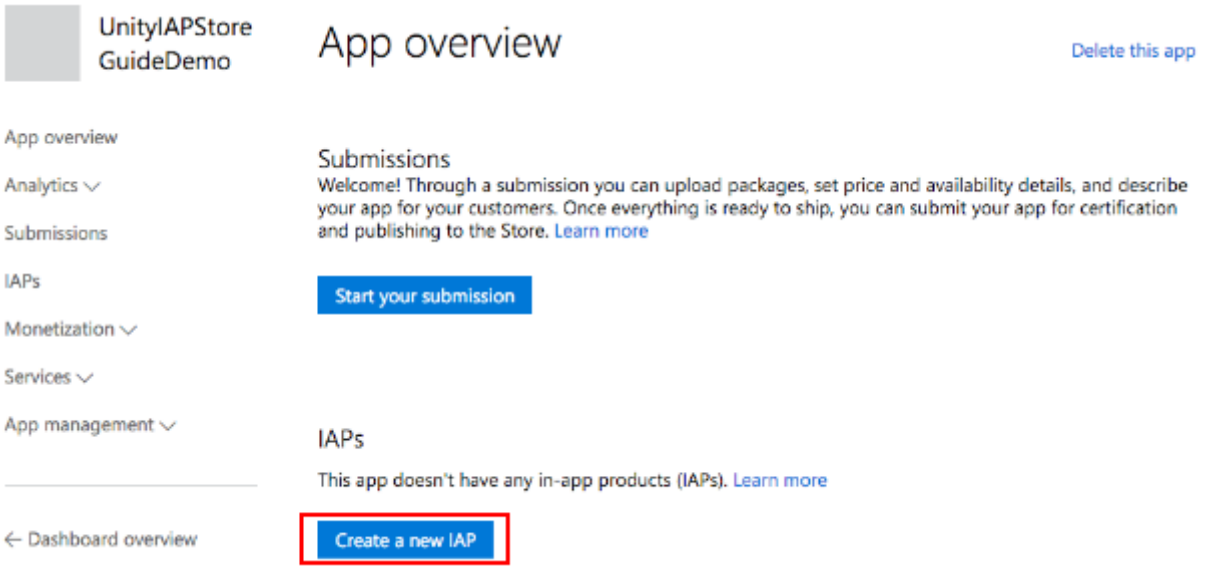
Submit the app for Certification.

Submissions may take many hours to complete, and blocking issues may be raised by Microsoft Certification, which you will need to address before the submission passes successfully.

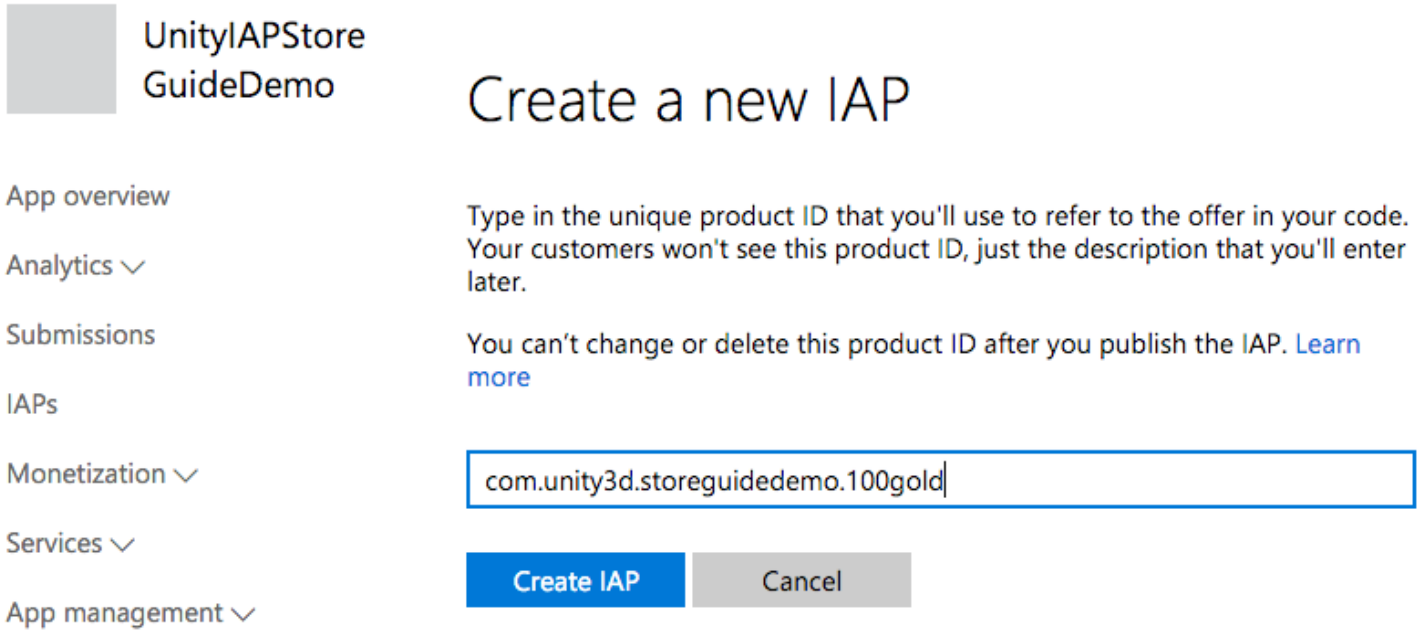
Add In-App Products on the Store

Add each IAP, setting the price to be “free” so that no money will be transacted during testing. After the test is completed, reconfigure the IAP with the desired price and republish it. See IAP Submissions.

In the new app’s “App overview” page, click **Create a new IAP** .



Enter the product ID.



Configure the type, price, and language.

NOTE: For **Pricing and availability** choose **free** for testing purposes to avoid incurring unnecessary financial charges. When you’re finished with testing, update and re-submit each IAP with the desired price in preparation for release to the public.

Select **Properties** to set the type.

Properties

Product type*

Consumable

You can't change product type after this IAP is published.

Content type*

Electronic software dowı

Keywords

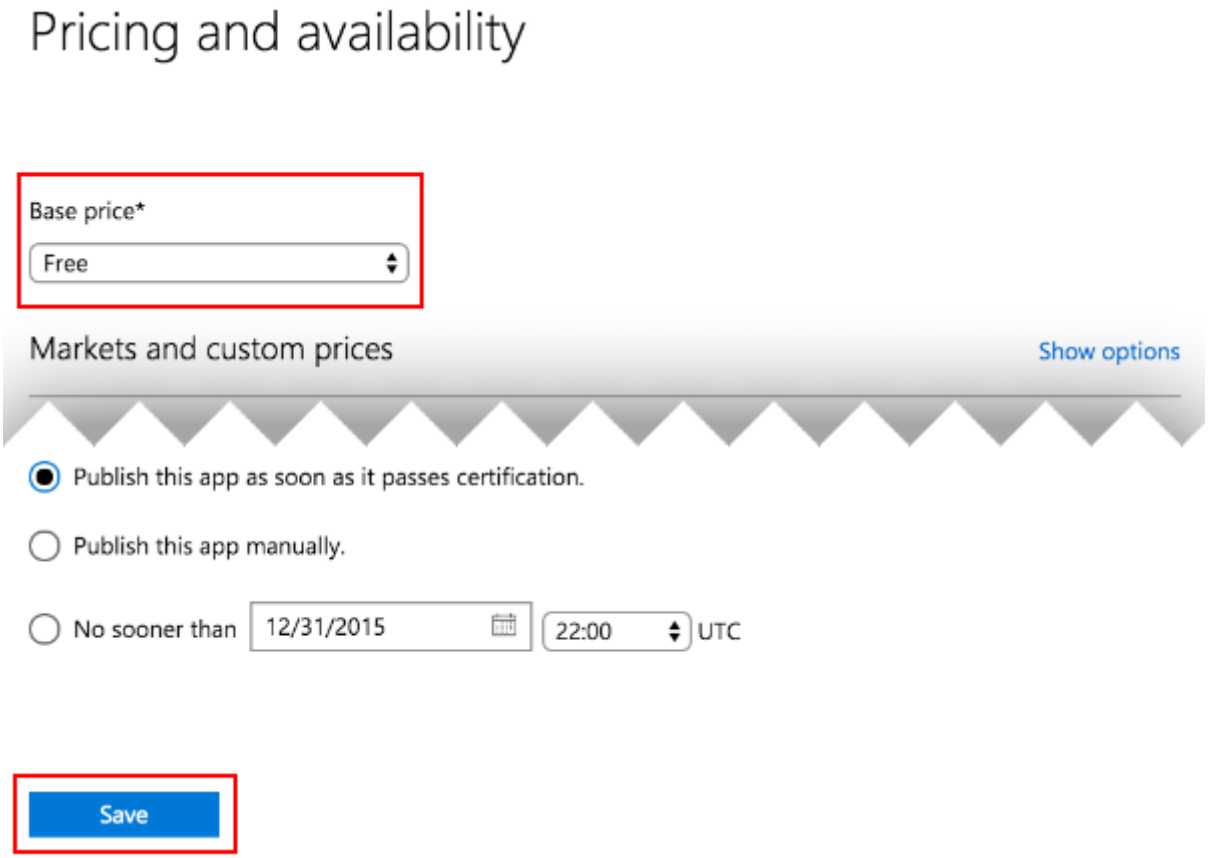
- Enter up to ten keywords to help categorize IAPs in your app. Learn more
- 30 character limit each
- Not available to packages targeting Windows 8 and Windows 8.1

Tag

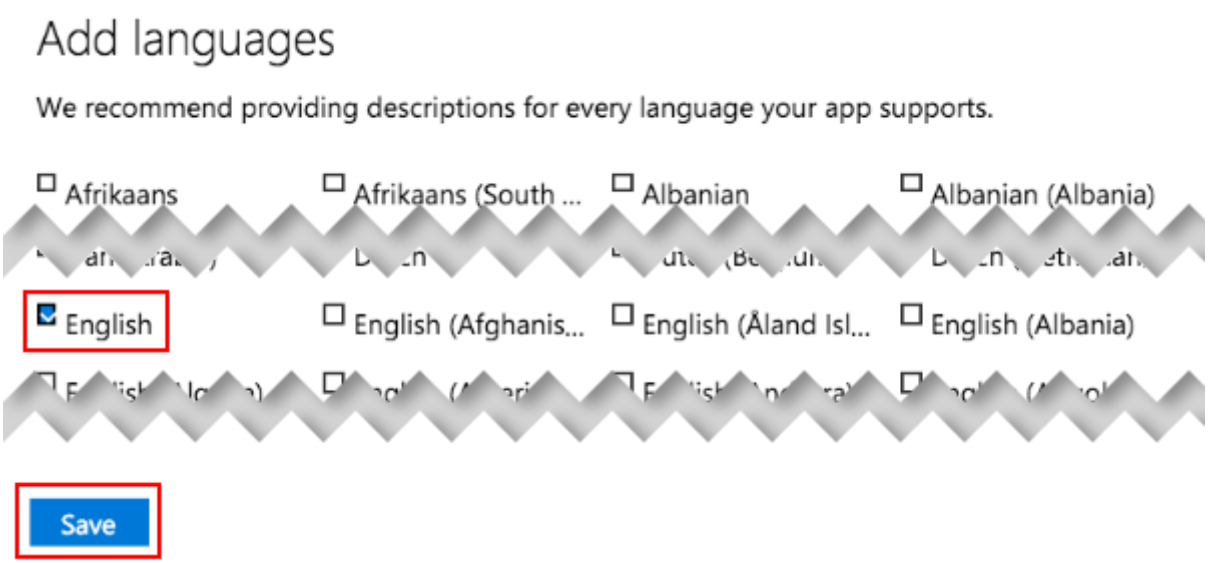
- Enter any custom information your app needs for this IAP. Learn more
- 3000 character limit
- Not available to packages targeting Windows 8 and Windows 8.1

Save

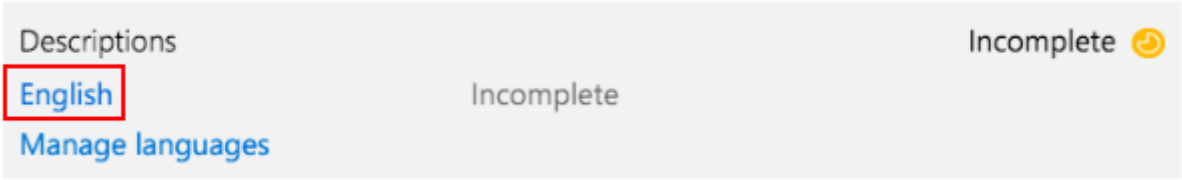
Select **Pricing and availability** to set the price choosing **Free** as explained above.



Select **Manage languages** and declare the supported languages.



Select the declared language when returned to the IAP overview.



Populate the Title, Description and Icon.

Description: English

Title*

100 Gold Coins

- 100 character limit

Description

A pile of lustrous gold coins

- 200 character limit

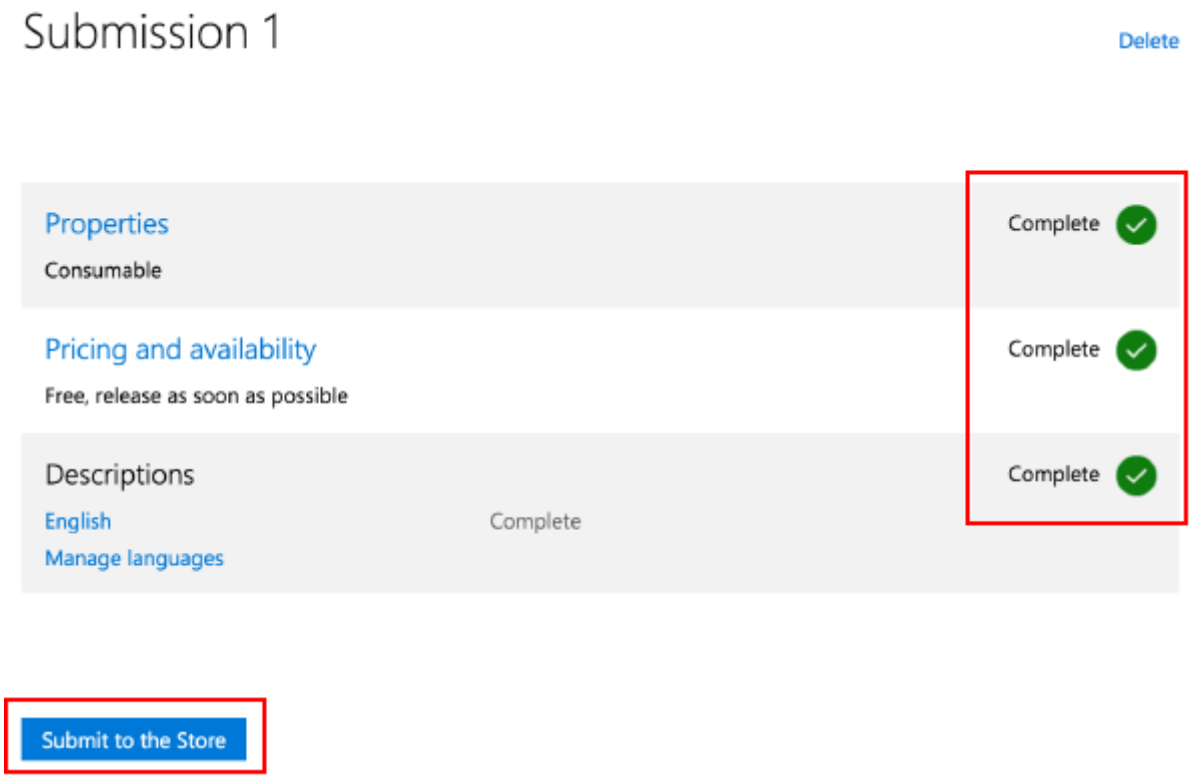
Icon

- 300 x 300 pixels
- Accepted file types: .png

Save

Submit the IAP for Certification.

As with apps, IAP submissions may take many hours to complete, and blocking issues may be raised by Microsoft Certification which you will need to address before the submission passes successfully.



Test IAP with the Store

These steps follow a branch of the beta test process made possible with Windows Store. This involves limiting the visibility of the app itself, negating discovery by the public through the “Search Store” function. See Windows Dev Centre resources on beta testing and targeted distribution for more information.

Confirm both the app and IAPs have completed Certification.

Install the app on a Windows 10 device via the direct link, generated above.

Test IAP.

After passing test, update the IAP with the desired public pricing, update the app visibility settings to share with the general public, and submit both kinds of changes for final Certification.

• 2017–05–16 Page amended with no editorial review

Configuration for the Amazon Appstore

Leave feedback

Introduction

This guide describes the process of setting up the Amazon Appstore for use with the Unity in-app purchasing (IAP) system. This includes establishing the digital records and relationships that are required to interact with the **Unity IAP** API, setting up an Amazon developer account, and testing and publishing a **Unity IAP** application.

As with other platforms, the Amazon store allows for the purchase of virtual goods and managed products. These digital products are identified using a string identifier and an additional type to define durability, with choices including subscription (capable of being subscribed to), consumable (capable of being rebought), and non-consumable (capable of being bought once).

Cross-store implementation of in-app purchases

There are cross-store installation issues with publishing to multiple Android IAP stores (e.g. Amazon and Google) simultaneously and shared Android bundle identifiers. Please see the page on Cross-store installation issues with Android in-app purchase stores to learn more.

Amazon Appstore

Getting started

Set up an Amazon developer account at the Amazon developer portal.
Write a game implementing the Unity IAP API. For reference, see the guides on Unity IAP initialization and Integrating Unity IAP in your game. Use the Amazon Appstore for apps with no restrictions on IAP items.

Device setup

For non-FireOS Android devices, download and install the Amazon Appstore.

On FireOS devices, the Amazon Appstore should come pre-installed.

Once you have installed the Amazon Appstore, install the Amazon App Tester.

Set up the Android SDK
To install and watch the Android debug log, ensure you have the Android SDK installed. Download the relevant command line tools package from the Android SDK install page and extract them to your computer.
Confirm that the SDK recognizes the attached Android device through the command-line **adb** tool. For example:

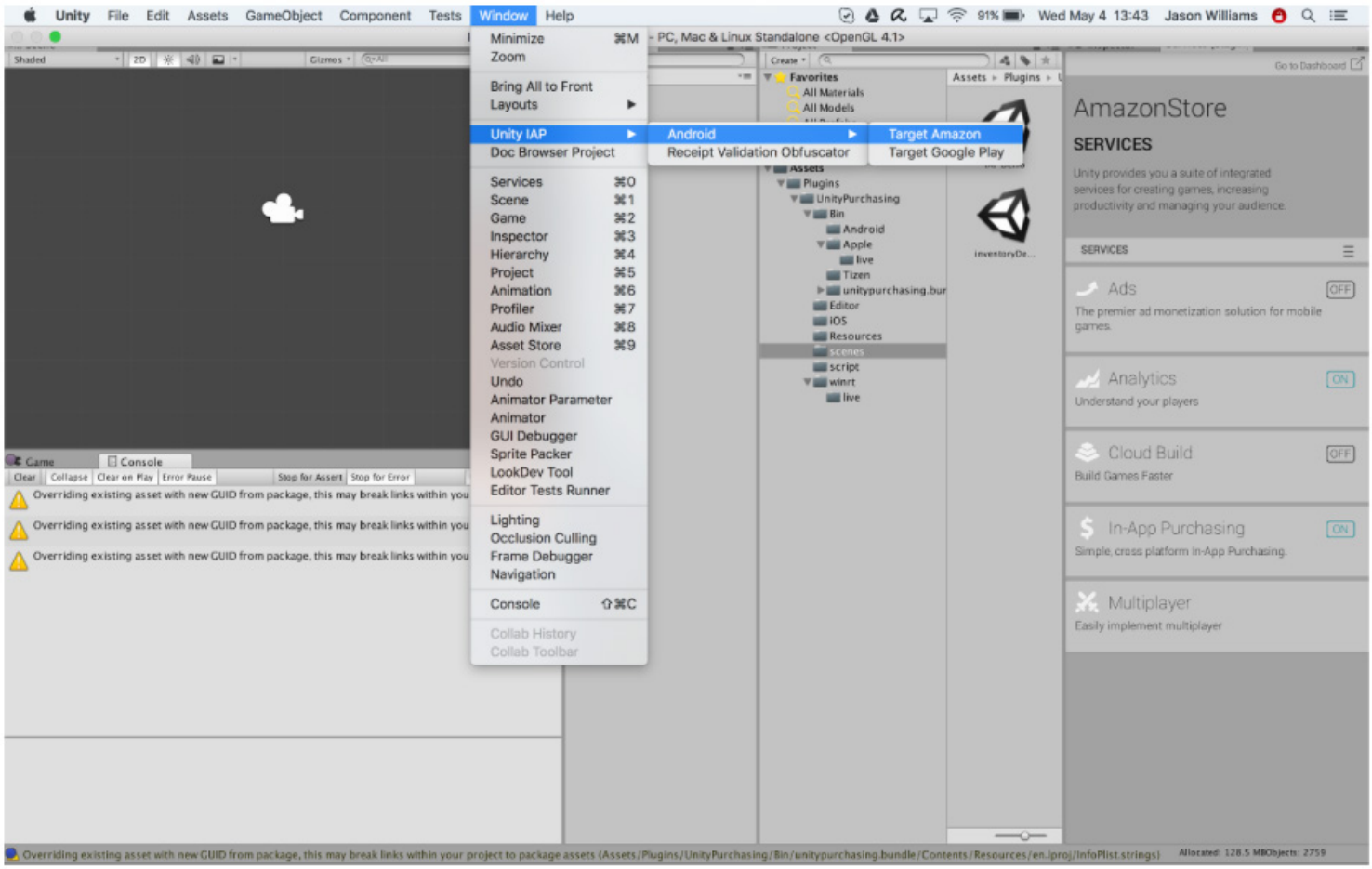
```
|[11:07:01] user@laptop:/Applications | $ adb devices
List of devices attached
00DA0807526300W5    device
```

Unity app setup

Setting up to use Unity's IAP takes a few steps.

Import the Unity IAP plug-in. See Setting up Unity IAP for more information (Unity 5.3 or higher).

Set the IAP target store. You should already have an Android app set up. Set the target store using **Unity IAP's Window > Unity IAP > Android > Target Amazon** menu item. This is used to toggle between Google, Amazon, and other Android stores.



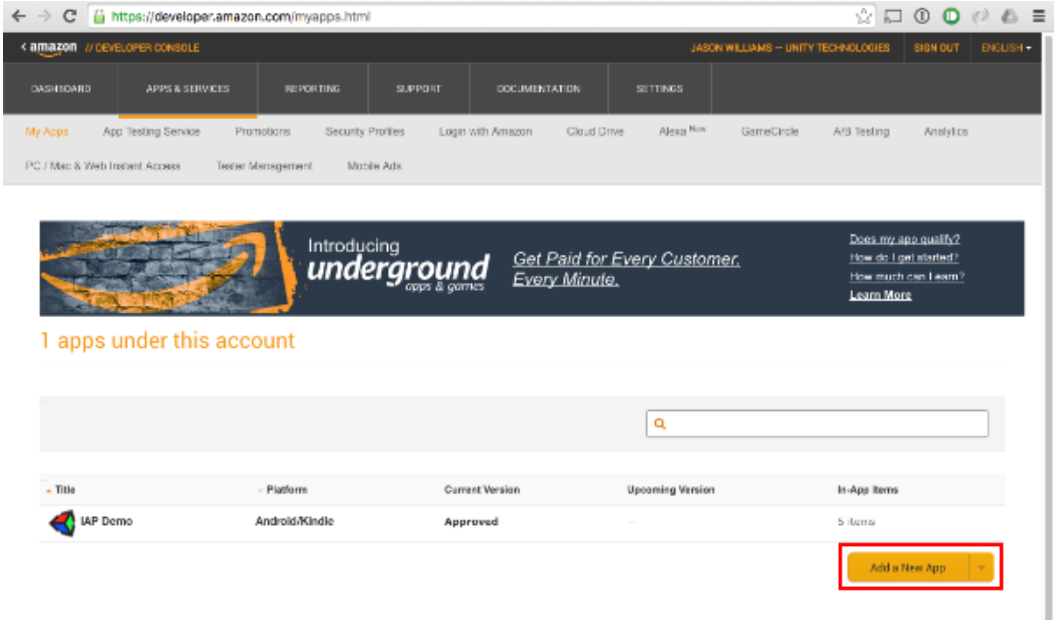
Alternatively, call the API:

```
UnityPurchasingEditor.TargetAndroidStore(AndroidStore.AmazonAppStore)
```

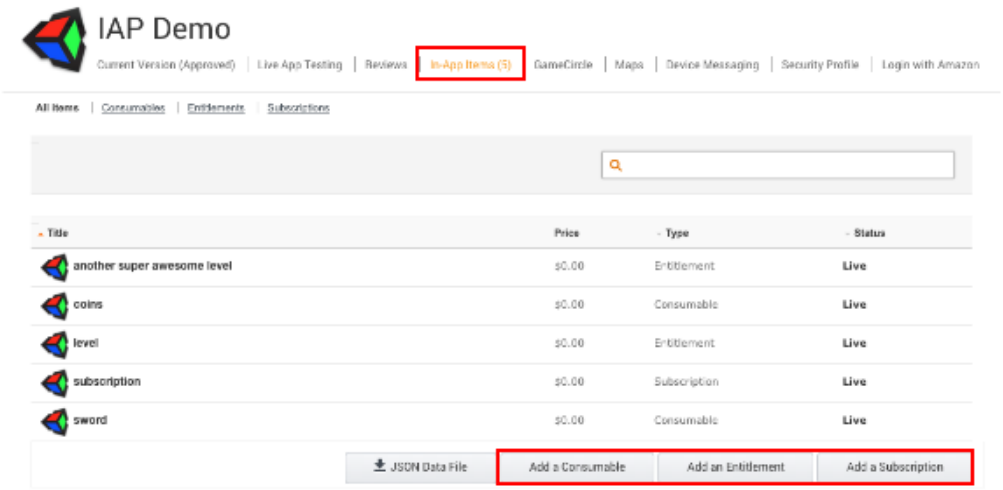
Amazon Appstore setup

It's not necessary to download Amazon's native IAP plug-in when preparing to use the Amazon stores, as all of the functionality it provides is already included in Unity's IAP service.

Add your app. From the Amazon Developer Portal select **Add a New App**.



Set up your catalog. Using the product descriptions you prepared earlier, add the items to the Amazon catalog using the Amazon Developer Portal. Navigate to your app's page, and find the **In-App Items section**. Use the **Add a Consumable**, **Add an Entitlement**, or **Add a Subscription** buttons to set up your catalog.



Samsung Galaxy IAP configuration

Leave feedback

This guide describes the process of establishing the digital records and relationships necessary for a Unity app to interact with an in-app purchase store. The Unity IAP purchasing API is targeted.

In-app purchasing (IAP) is the process of transacting money for digital goods. A platform's store allows the purchase of products, representing digital goods. These products have an identifier, typically of string datatype. Products have types to represent their durability: the most common are subscription (capable of being subscribed to), consumable (capable of being rebought), and non-consumable (capable of being bought once).

Cross-store implementation of in-app purchases

Note that there are cross-store installation issues when using shared Android bundle identifiers to publish to multiple Android in-app purchase stores (such as Samsung and Google) simultaneously. See documentation on Cross-store installation issues with Android in-app purchasing for more information.

Samsung Galaxy apps

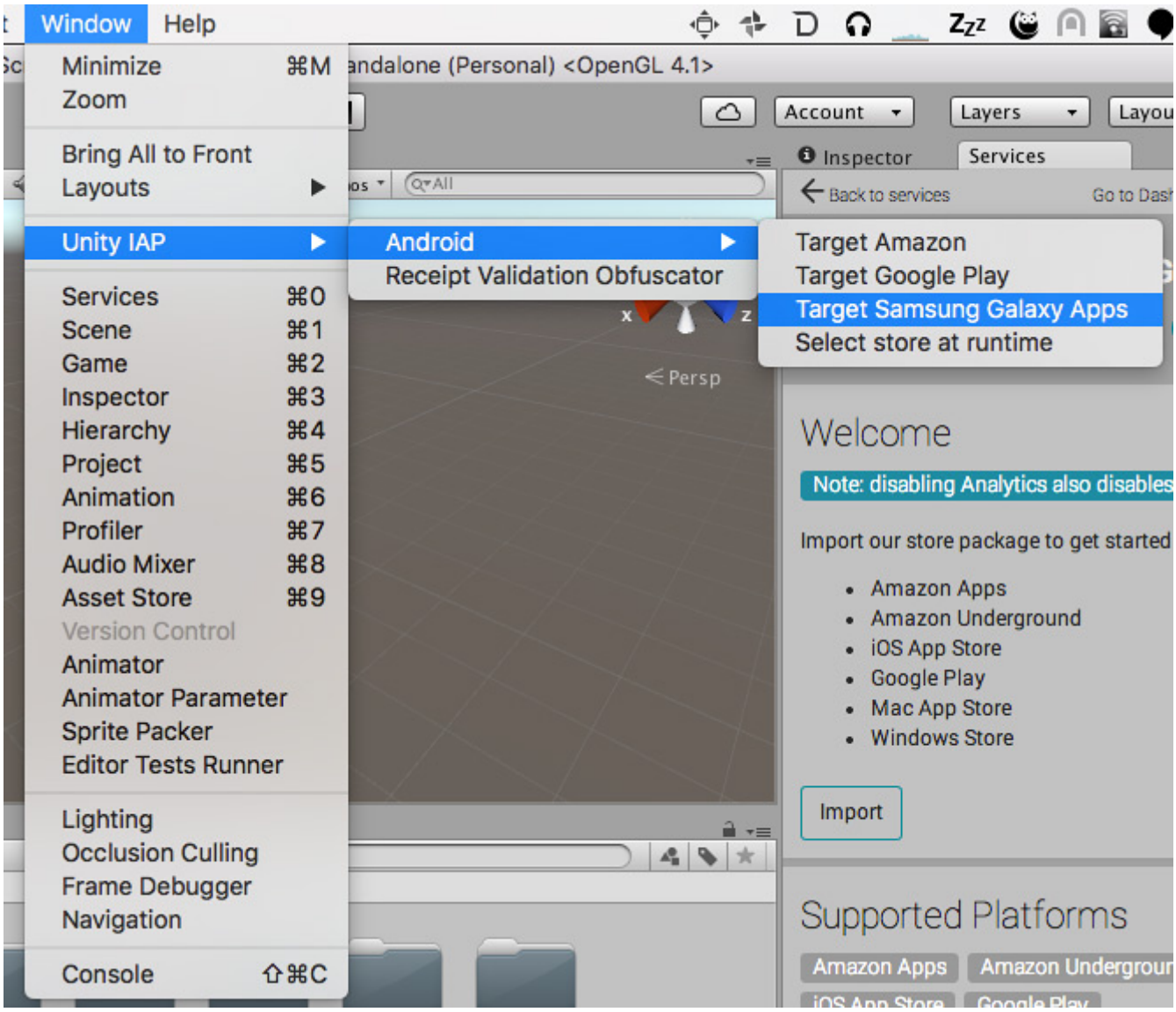
Getting started

Write an app implementing Unity IAP. See Unity IAP initialization and Integrating Unity IAP with your app.

Keep the app's product identifiers on-hand for use with the Samsung Apps Seller Office later.

```
public class MyStoreClass : MonoBehaviour, IStoreListener {
    void Start() {
        var module = StandardPurchasingModule.Instance();
        ConfigurationBuilder builder = ConfigurationBuilder.Instance(module);
        builder.AddProduct("com.unity3d.storeguidedemo.100gold", ProductType.Consumable);
        UnityPurchasing.Initialize(this, builder);
    }
```

To set the IAP target store in the Unity Editor, go to **Window** > **Unity IAP** > **Android** > **Target Samsung Galaxy Apps**.



Alternatively, call the Editor API:

```
UnityPurchasingEditor.TargetAndroidStore(AndroidStore.SamsungApps
```

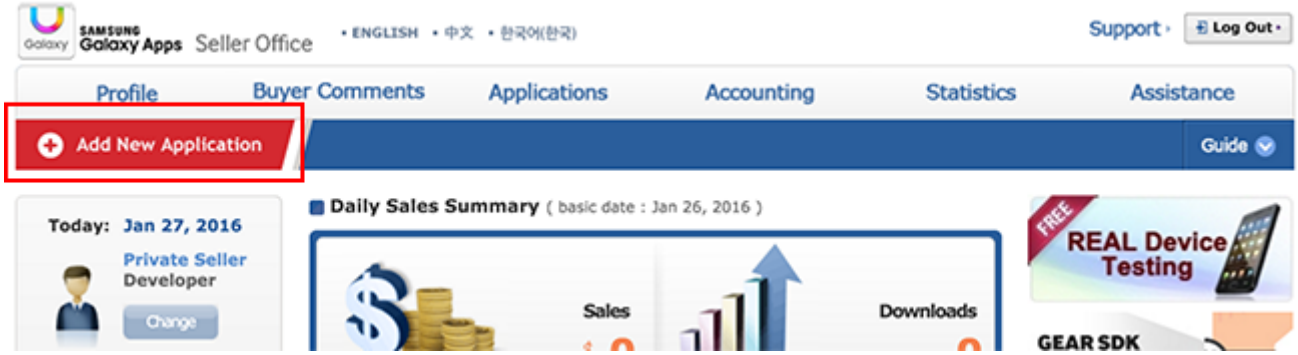
Build a signed non-Development Build Android APK from your app. See documentation on Getting started with Android development to learn more.

Tip: Take special precautions to safely store your keystore file. The original keystore is always required to update a published application.

Register the application

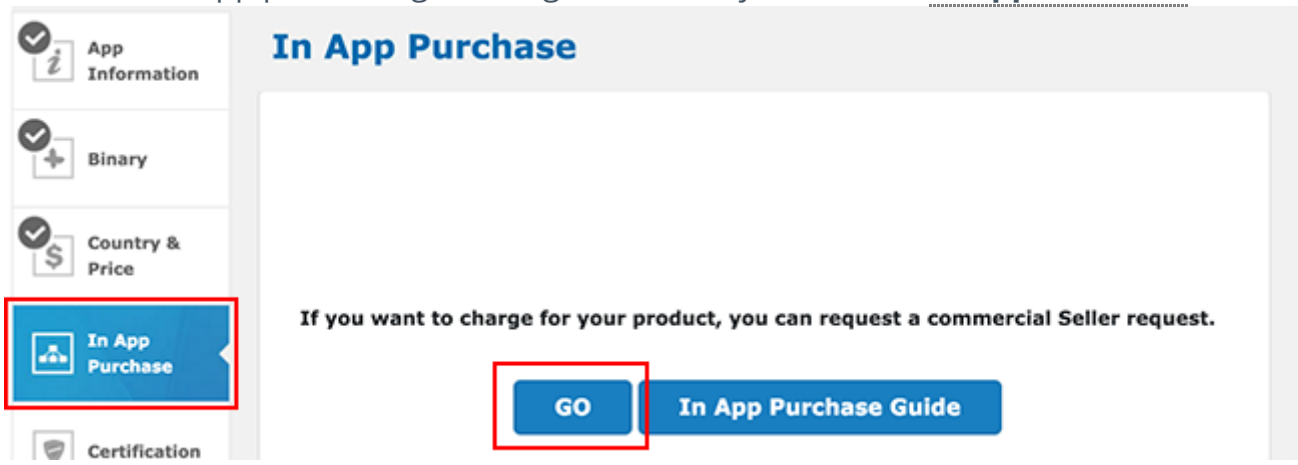
Register the Android application with the Samsung Galaxy Apps Seller Office.

Choose **Add New Application**.

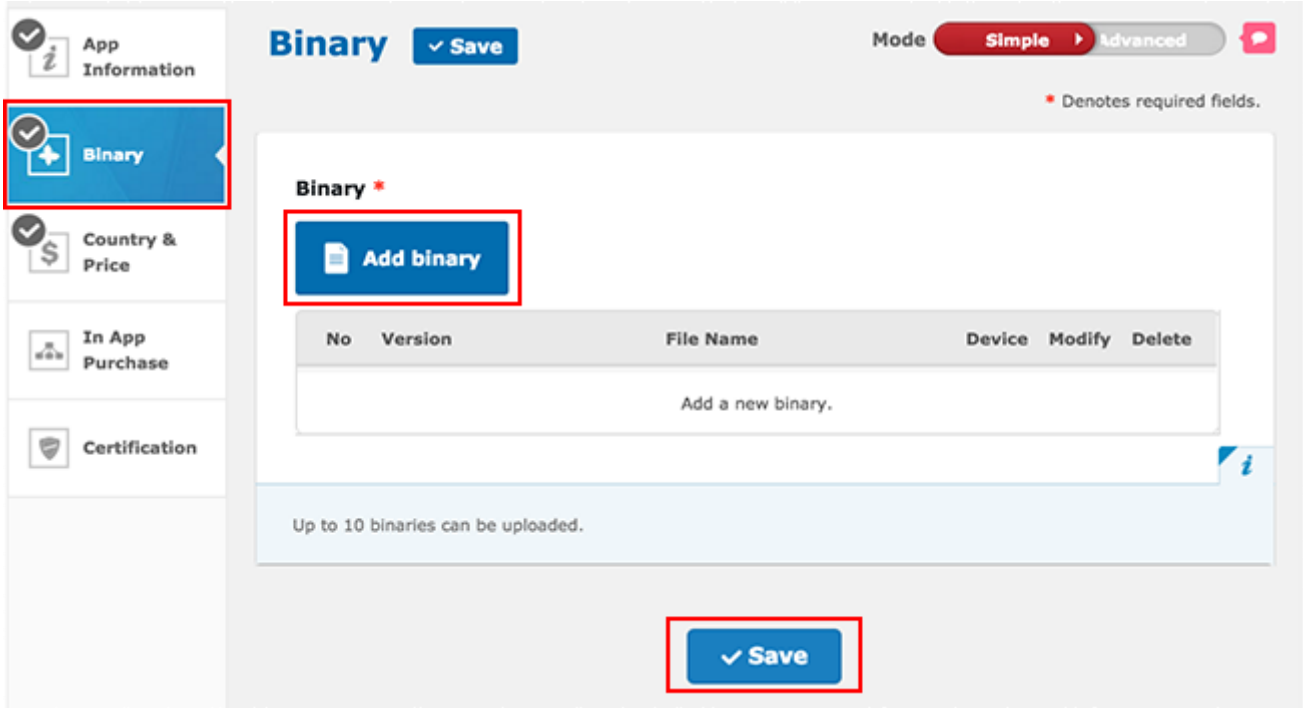


Choose the **Android** option and select a **Default Language**.

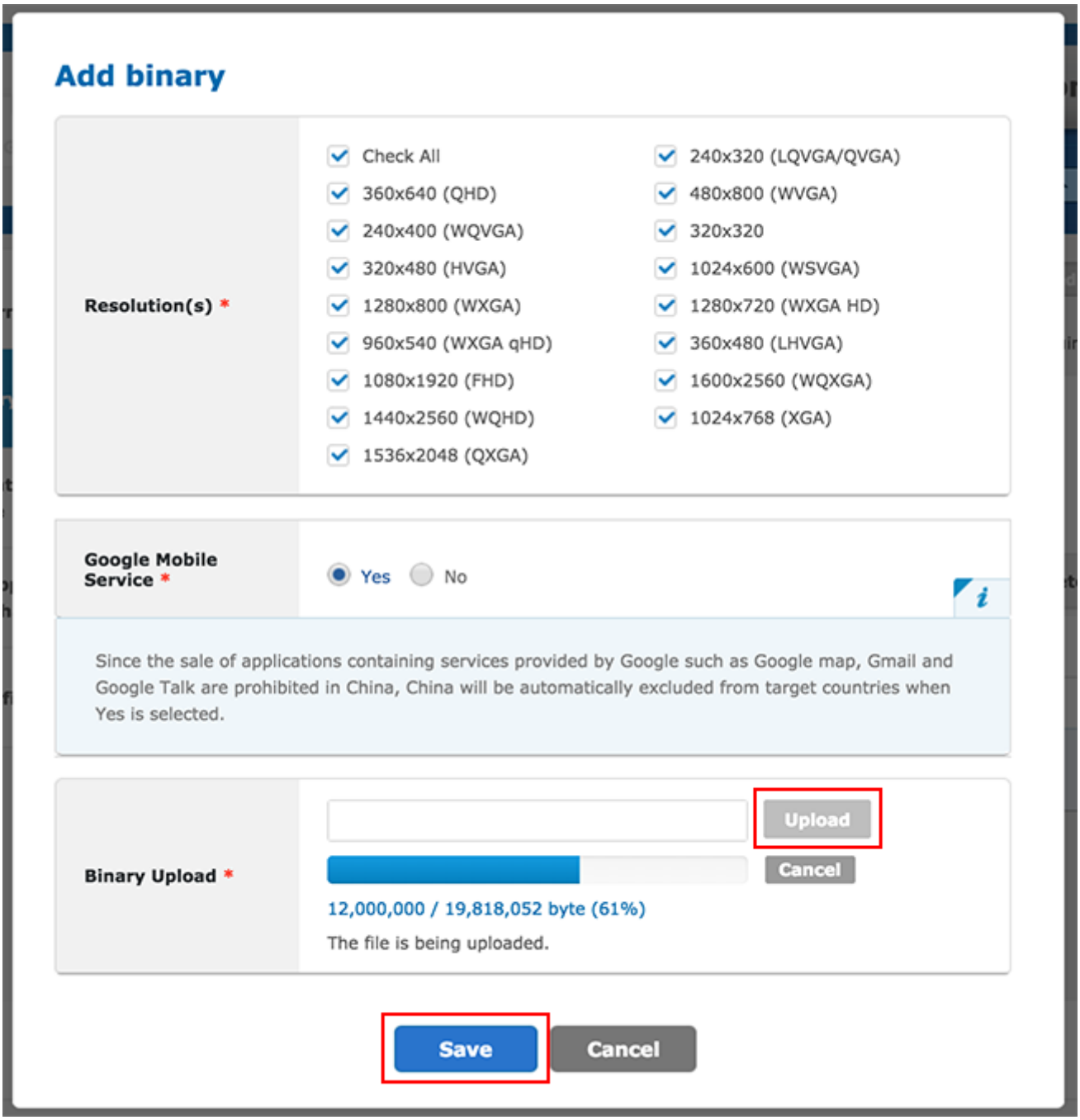
To enable in-app purchasing, first register a binary APK. Go to **In App Purchase** and click **GO**.



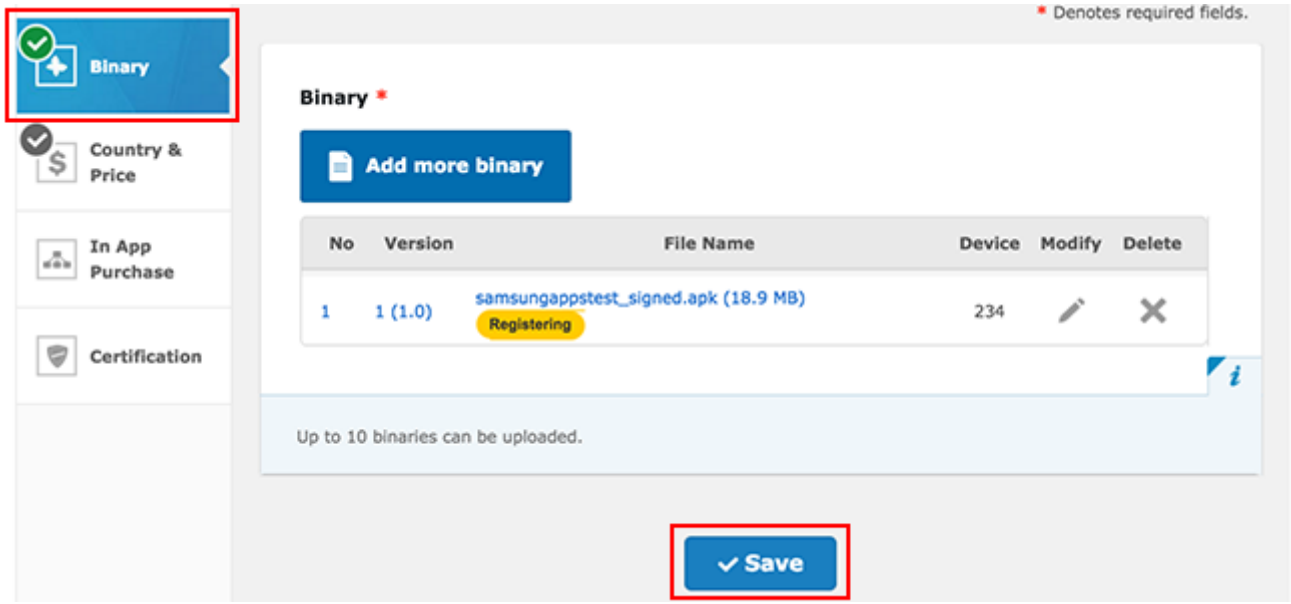
In the App Store Developer Console, go to **Binary** and select **Add binary**.



Populate the device characteristics in **Resolution(s)** and **Google Mobile Service**, upload your APK (the one you created above in the “Getting Started” section) in **Binary upload**, then click **Save**.



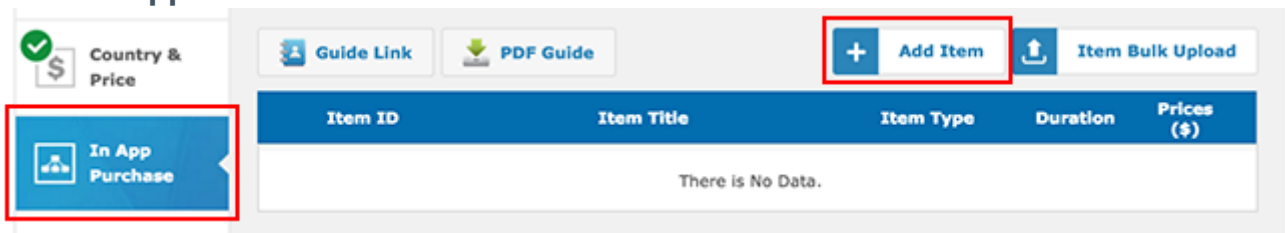
Wait for the APK upload to complete, then click **Save**.



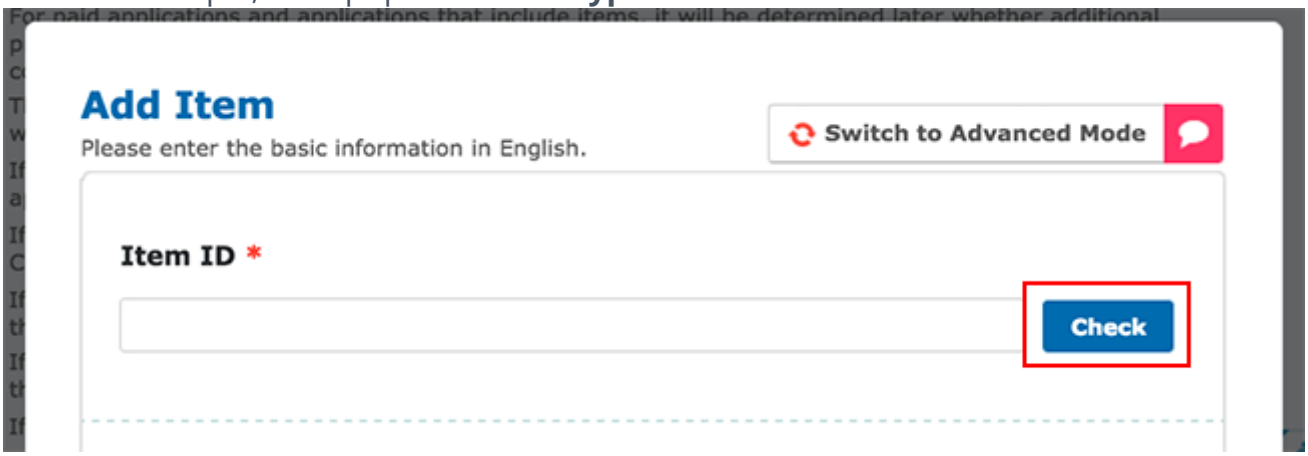
Add in-app purchases

In the Seller Office, add one or more in-app purchases for the app.

Go to **In App Purchase** and choose **Add Item**.



Define the **Item ID**. The Item ID here is the same identifier used in the app source code, added to the Unity IAP ConfigurationBuilder instance via `AddProduct()` or `AddProducts()`. For debugging purposes, it's best practise to use reverse-DNS for your Item ID. Click **Check** to ensure the Item ID is valid and unique, then populate **Item Type** and all other elements and click **Save**.



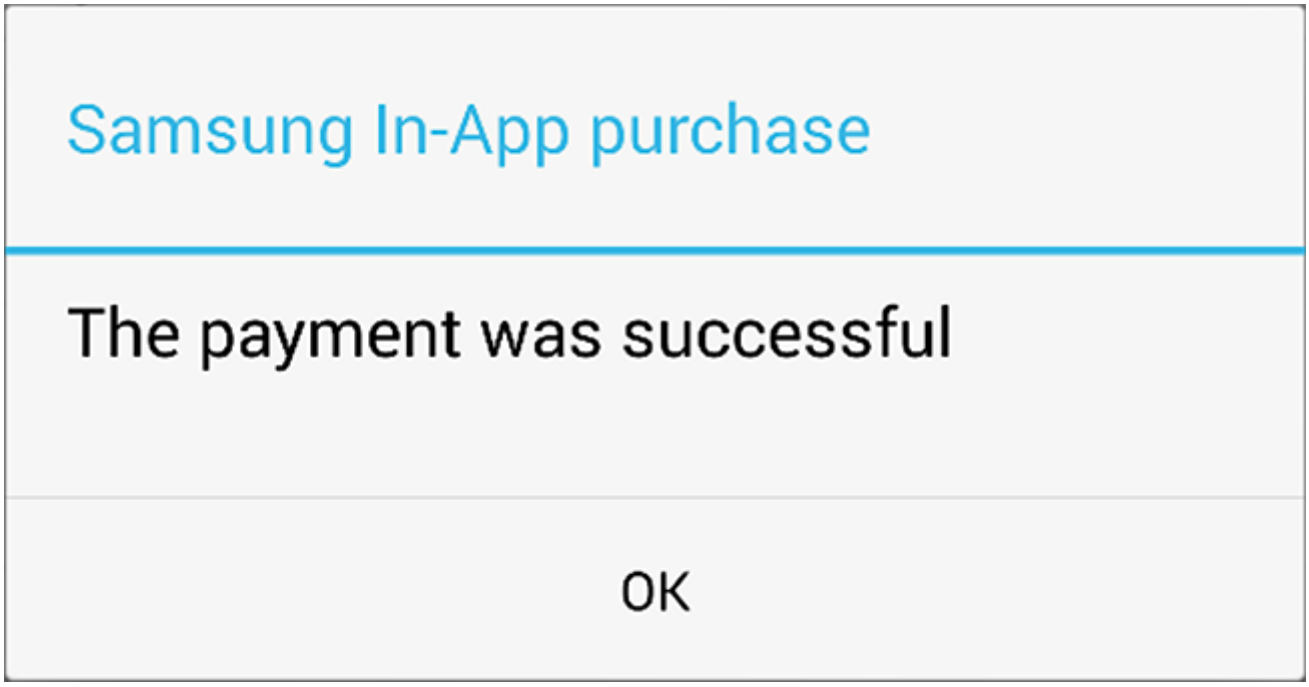
View the result in **In App Purchase**:

Testing an IAP implementation

The Samsung Galaxy App Store supports testing via the **Developer mode** value in the app before making purchases. This special build of the app connects with Samsung's billing servers and performs fake purchases. This does not incur real-world monetary costs related to the product, and allows you to test the app's purchasing logic.

Modify the app's Unity IAP integration, adding the following line after creating the `ConfigurationBuilder` instance: `builder.Configure<ISamsungAppsConfiguration>().SetMode(SamsungAppsMode.AlwaysSucceed); // TESTING: auto-approves all transactions by Samsung`. You can also configure this to fail all transactions via the `SamsungAppsMode.AlwaysFail` enumeration, enabling you to test all your error code.

Build and run the app, testing its in-app purchasing logic. As long as developer mode is implemented, this does not incur real-world monetary costs.



IMPORTANT: When testing is complete, make sure you remove the `SetMode` line. This ensures users pay real-world money when tha app is in use.

Codeless IAP

Leave feedback

Codeless IAP is the easiest way to implement in-app purchases in your Unity app. The Unity Editor offers an interface for configuring basic IAP integration using minimal script writing.

Codeless IAP handles the actual IAP transaction without requiring any code. Implementing Codeless IAP is a two-step process using the Editor:

Add **IAP Buttons** to your game.
Define your **Products** in the **IAP Catalog** .
The **Unity Purchasing** system configures the Products you populate in the catalog at run time. When players select an **IAP Button**, it initiates the purchase flow for the associated Product.

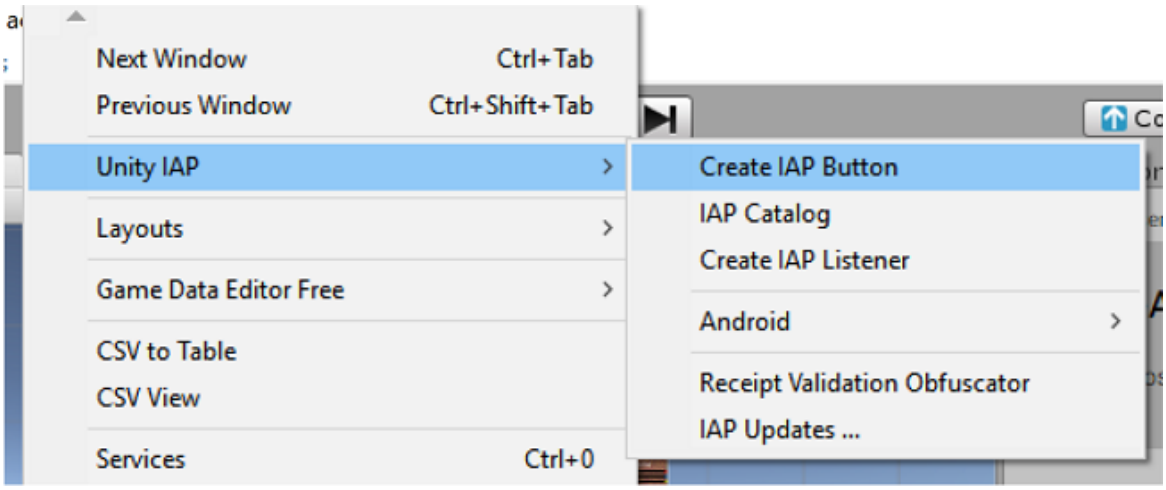
Note: You still need to use scripting to define how players access their newly purchased content. For more information, see the **Purchase fulfillment** section below.

Implementing Codeless IAP

Before starting, install the latest **Unity IAP** SDK. See documentation on **Setting up Unity IAP** for more information.

Adding IAP Buttons to your Scene

To add an **IAP Button** to your Scene, in the Unity Editor, select **Window > Unity IAP > Create IAP Button**.

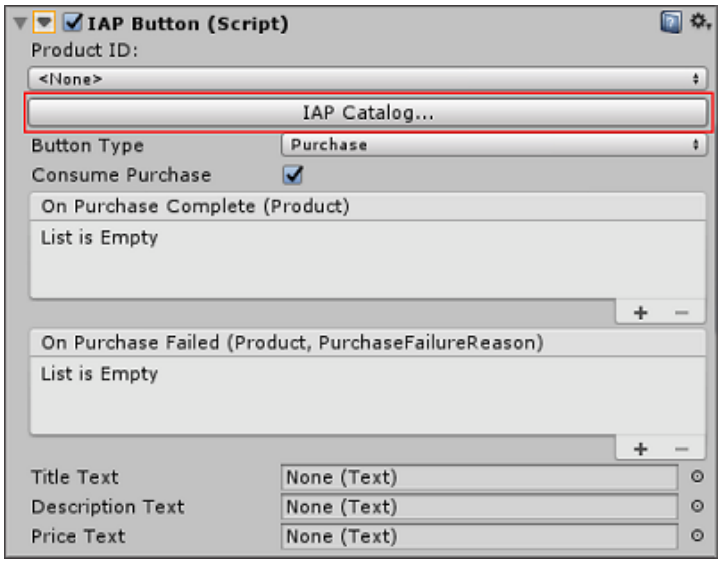


Creating a Codeless **IAP Button** in the Unity Editor

Populating Products in the IAP Catalog

Open the **IAP Catalog** GUI one of two ways:

Select **Window > Unity IAP > IAP Catalog**.
Or, with your **IAP Button** selected, locate its **IAP Button (Script) component** in the **Inspector**, then click **IAP Catalog...**.

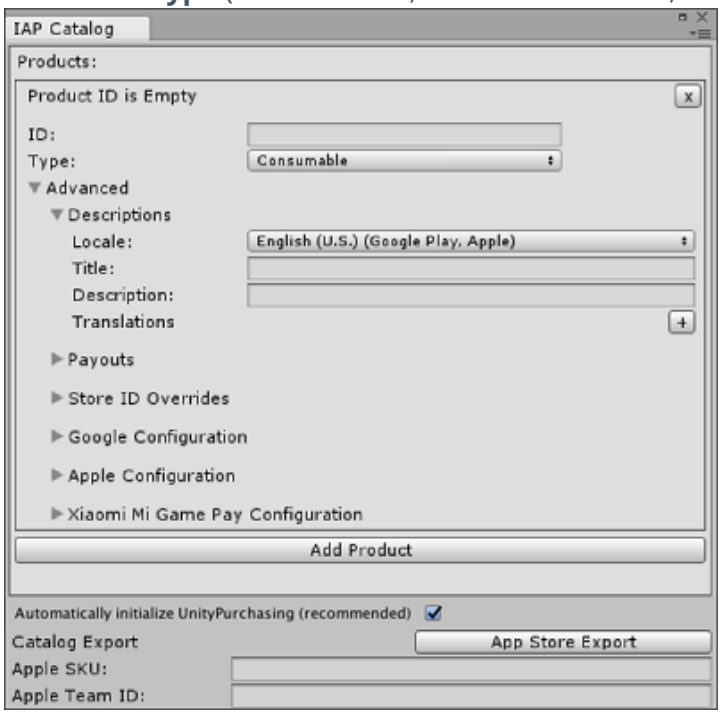


Accessing the **IAP Catalog** GUI through an **IAP Button** script component

Next, use the GUI to define the following attributes for each Product in your catalog.

A **Product ID** with which to communicate to app stores (note that you can override this ID with unique store-specific IDs through the **Advanced** option).

A **Product Type** (**Consumable**, **Non-Consumable**, or **Subscription**).

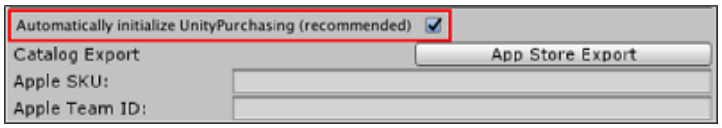


Populating Product information in the **IAP Catalog** GUI

Note: The **IAP Catalog** GUI provides additional tools for configuring your Products. Before exporting a catalog for upload to its respective store, you must populate description and pricing information as well. For detailed information on these settings, see documentation on **Defining Products** .

Automatically initializing `UnityPurchasing`

The IAP SDK must initialize in order for in-app purchasing to work. This occurs automatically when the first instance of a Codeless **IAP Button** or **IAP Listener** loads at run time. However, you may need to initialize the SDK before an IAP Button or IAP Listener appears in your game (for example, serving an **IAP Promo** offer after application launch). In these cases, check **Automatically initialize UnityPurchasing (recommended)** at the bottom of the **IAP Catalog** window. This ensures that `UnityPurchasing` initializes immediately when the application starts, and eliminates dependencies on the codeless instances' lifecycles.



Enabling auto-initialization for the SDK through the **IAP Catalog** GUI

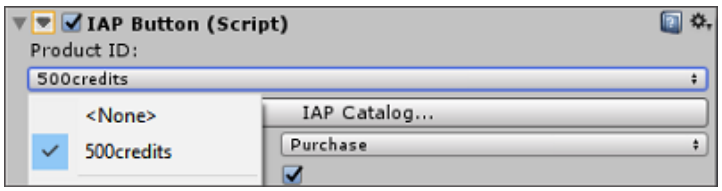
In order to work, your catalog must contain at least one Product.

Note: You can use auto-initialize together with IAP Buttons or Listeners. In this case, the SDK initializes when the game starts instead of when the first instance of an IAP Button or Listener loads in the **Scene**. However, you should not enable auto-initialize if you also initialize manually in a script, as this may cause errors.

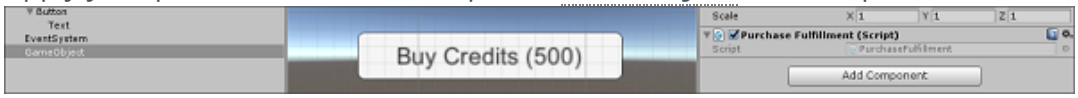
Purchase fulfillment

When your catalog contains at least one Product, you can define **IAP Button** behavior when the purchase completes or fails.

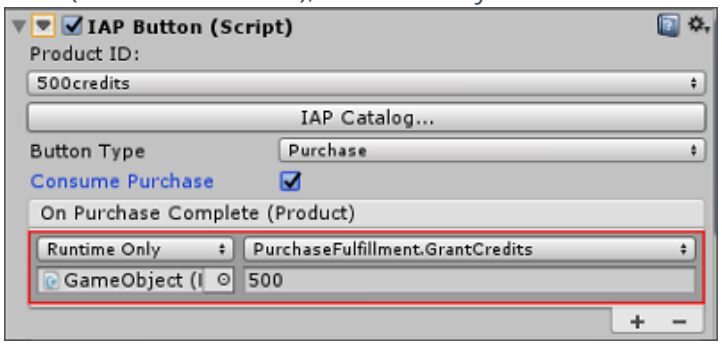
Select your **IAP Button** in the **Scene view**, then locate its **IAP Button (Script)** component in the Inspector.
Select the Product to link to the **IAP Button** from the **Product ID** drop-down list.



Create your own function that provides purchase fulfillment, or import an **Asset** that does this (see code sample, below).
Apply your purchase fulfilment script to a **GameObject** as a component.



Return to the **IAP Button (Script)** component in the Inspector, and click the plus (**+**) button to add a function to the **On Purchase Complete (Product)** list.
Drag the GameObject with the purchase fulfillment script onto the **On Purchase Complete (Product)** event field (illustrated below), then select your function from the dropdown menu.



Fulfillment script code sample:

```
public void GrantCredits (int credits){
    userCredits = userCredits + credits;
    Debug.Log("You received " + credits " Credits!");
}
```

Run your game to test the **IAP Button**.

Extended functionality

Exporting to an app store

In order for purchases to function, you must configure your catalog on the corresponding app store. To do so, you can export your Product Catalog as a CSV file to Google Play, or as an XML file through Apple's Application Loader to the iTunes Store.

Google Play

To export your Product Catalog for Google Play:

In the **IAP Catalog** window (**Window > Unity IAP > IAP Catalog**), ensure that each Product has the following defined:
* **ID**
* **Type**
* **Title** and **Description**
* **Price** or **Pricing Template** ID
Scroll down, and select **App Store Export**.
Select **Google Play CSV**.
Choose a location in which to save the CSV file.
For complete guidance on uploading your exported catalog to Google Play, see the Google in-app billing documentation on the Android Developers website.

Apple iTunes

To export your Product Catalog for Apple iTunes:

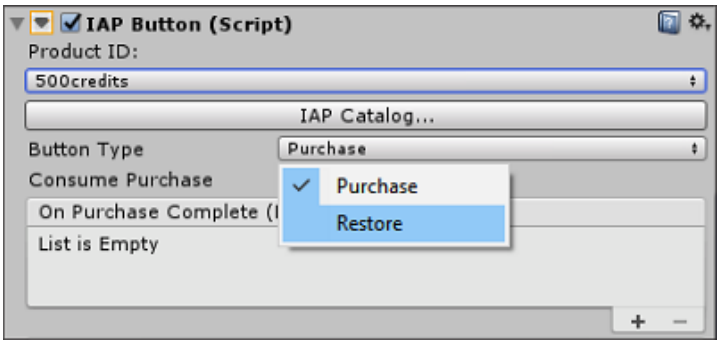
In the **IAP Catalog** window (**Window** > **Unity IAP** > **IAP Catalog**), ensure that each Product has the following defined:
* **ID**
* **Type**
* **Title** and **Description**
* **Price Tier** and **Screenshot path**
* **Apple SKU** (found in iTunes Connect)
* **Apple Team ID** (found on Apple's developer website)
Scroll down, and select **App Store Export**.
Select **Apple XML Delivery**.
Choose a location in which to save the XML file.
For complete guidance on importing through Apple's Application Loader, see the Application Loader documentation on the iTunes Connect website.

Restore Button

Some app stores, including iTunes, require apps to have a **Restore** button. Codeless IAP provides an easy way to implement a restore button in your app.

To add a **Restore** button:

Add an **IAP Button** to your Scene (**Window** > **Unity IAP** > **Create IAP Button**).
With your **IAP Button** selected, locate its **IAP Button (Script)** component in the Inspector, then select **Restore** from the **Button Type** drop-down menu (most of the component's other fields will disappear from the

Inspector view). 

When a user selects this button at run time, the button calls the purchase restoration API for the current store. This functionality works on the **iOS** App Store, the Mac App Store, the Windows Store, Samsung Apps, and the Cloud Moolah app store. You may want to hide the **Restore** button on other platforms.

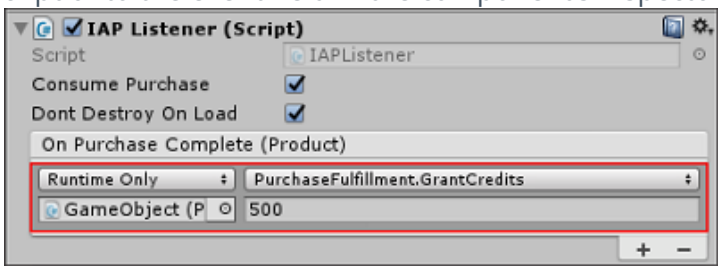
If the restore succeeds, **Unity IAP** invokes the **On Purchase Complete (Product)** function on the **IAP Button** associated with that Product.

For more information, see the documentation on **Restoring purchases** .

IAP Listeners

Codeless IAP dispatches successful and failed purchase events to an active **IAP Button** component in the hierarchy. However, there may be times when it is difficult or undesirable to have an active **IAP Button** when handling a successful purchase. For example, if a purchase is interrupted before completing, Unity IAP attempts to process the purchase again the next time it is initialized. You may want this to happen immediately after the app launches, in which case an **IAP Button** wouldn't make sense. Codeless IAP includes the **IAP Listener** component precisely for these cases. An active **IAP Listener** in the Scene hierarchy receives any purchase events that cannot be dispatched to an **IAP Button**.

To add an **IAP Listener**:

In the Unity Editor, select **Window > Unity IAP > Create IAP Listener**.
Follow the steps for writing a purchase fulfillment script as a GameObject component.
Select the **IAP Listener** in the Scene and locate its **IAP Listener (Script)** component in the Inspector, then click the plus (**+**) button to add a function to the **On Purchase Complete (Product)** list.
Drag the GameObject with the purchase fulfillment script onto the event field in the component's Inspector, then select your function from the dropdown menu. 

Accessing Unity IAP's extended functionality

The Codeless IAP feature does not expose most of Unity IAP's extended functionality. However, Codeless IAP is implemented on top of the existing scripting APIs, so you can modify much of its functionality in the *IAPButton.cs* script (*Assets/Plugins/UnityPurchasing/script/IAPButton.cs*) to suit your needs.

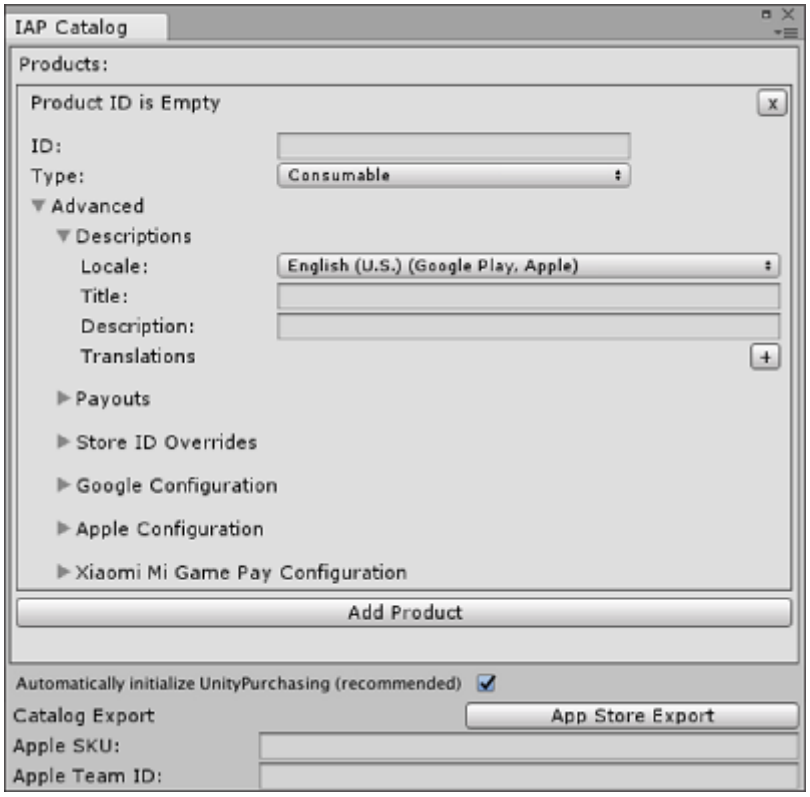
To use Unity IAP's extended functionality, access the Unity IAP `IStoreController` and `IExtensionProvider` instances returned by `IStoreListener.OnInitialize` .

2018–05–30 Page amended with editorial review

Defining products

Leave feedback

In order to use in-app purchases, your app must provide a list of Products for sale. You can do this through scripting, or using the **Codeless IAP Catalog** (**Window** > **Unity IAP** > **IAP Catalog**). Whichever implementation you use, you must define the appropriate attributes for each Product. This page covers these attributes in detail.



The **IAP Catalog** GUI in the Unity Editor

Product ID

Enter a cross-platform unique identifier to serve as the Product's default ID when communicating with an app store.

Important: The ID may only contain lowercase letters, numbers, underscores, or periods.

Product Type

Each Product must be of one of the following Types:

Type	Description	Examples
Consumable	Users can purchase the Product repeatedly. Consumable Products cannot be restored.	* Virtual currencies * Health potions * Temporary power-ups.
Non-Consumable	Users can only purchased the Product once. Non-Consumable Products can be restored.	* Weapons or armor * Access to extra content

Type	Description	Examples
Subscription	Users can access the Product for a finite period of time. Subscription Products can be restored.	* Monthly access to an online game * VIP status granting daily bonuses * A free trial

Note: For more information on Subscription type support, see the section on **Subscription Product support** .

Advanced

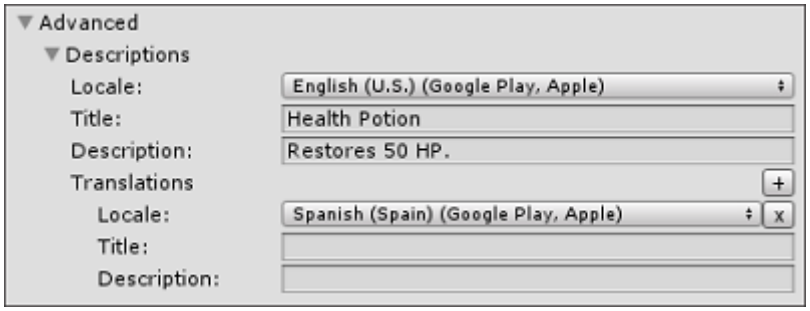
This section defines the metadata associated with your Product for use in an in-game store.

Descriptions

Use the following fields to add descriptive text for your Product:

Field	Data type	Description	Example
Product Locale	Enum	Determines the app stores available in your region.	**English (U.S.)** (Google Play, Apple)
Product Title	String	The name of your Product as it appears in an app store.	“Health Potion”
Product Description	String	The descriptive text for your Product as it appears in an app store, usually an explanation of what the Product is.	“Restores 50 HP.”

Add **Translations** for the **Title** and **Description** fields by clicking the plus (**+**) icon and selecting an additional locale. You can add as many translations as you like.



Populating **Descriptions** fields for Products in the **IAP Catalog** GUI

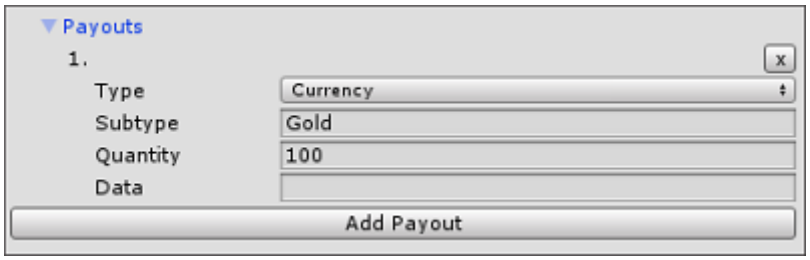
Payouts

Use this section to add local, fixed definitions for the content you pay out to the purchaser. Payouts make it easier to manage in-game wallets or inventories. By labeling a Product with a name and quantity, developers can quickly adjust in-game counts of certain item types upon purchase (for example, coins or gems).

Note: This functionality is only available in Unity 2017.2 or higher.

Field	Data type	Description	Example

Field	Data type	Description	Example
Payout Type	Enum	Defines the category of content the purchaser receives. There are four possible Types.	* Currency * Item * Resource * Other
Payout Subtype	String	Provides a level of granularity to the content category.	* “Gold” and “Silver” subtypes of a **Currency** type * “Potion” and “Boost” subtypes of an **Item** type
Quantity	Int	Specifies the number of items, currency, and so on, that the purchaser receives in the payout.	* 1 * >25 * 100
Data		Use this field any way you like as a property to reference in code.	* Flag for a UI element * Item rarity



Populating **Payouts** fields for Products in the **IAP Catalog** GUI

Note: You can add multiple Payouts to a single Product.

For more information on the PayoutDefinition class, see the Scripting Reference. You can always add Payout information to a Product in a script using this class. For example:

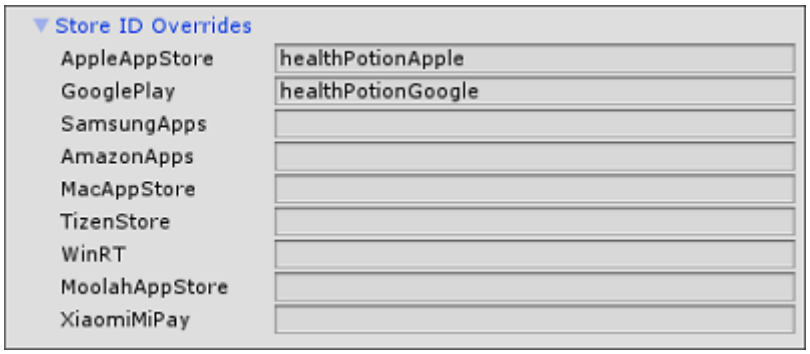
```
using UnityEngine.Purchasing;

new PayoutDefinition (PayoutType.Currency, "Gold", 100)
```

Note that the IAP Catalog acts as a Product catalog dictionary, not as an inventory manager. You must still implement the code that handles conveyance of the purchased content.

Store ID Overrides

By default, Unity IAP assumes that your Product has the same identifier (specified in the **ID** field, above) across all app stores. Unity recommends doing this where possible. However, there are occasions when this is not possible, such as when publishing to both **iOS** and Mac stores, which prohibit developers from using the same product ID across both.

In these cases, use the override fields to specify the Product’s correct identifier where it differs from the cross-platform ID.



Populating **Sore ID Overrides** fields for Products in the **IAP Catalog** GUI

You can also do this programmatically, as follows:

```
using UnityEngine;
using UnityEngine.Purchasing;

public class MyIAPManager {
    public MyIAPManager () {
        var builder = ConfigurationBuilder.Instance(StandardPurchasingModule.Ins
        builder.AddProduct("100_gold_coins", ProductType.Consumable, new IDs
        {
            {"100_gold_coins_google", GooglePlay.Name},
            {"100_gold_coins_mac", MacAppStore.Name}
        });
        // Initialize Unity IAP...
    }
}
```

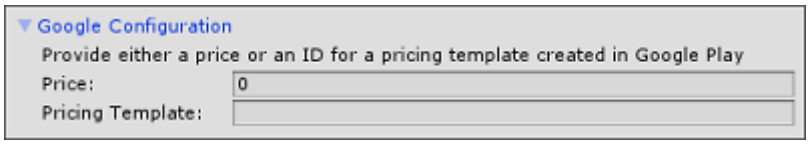
In this example, the Product identifies as “100_gold_coins_google” to Google Play and “100_gold_coins_mac” to the Apple App Store.

Note: Overrides only change the identifier Unity IAP uses when communicating with app stores. You should still use the Product’s cross-platform identifier when making API calls.

Important: The ID may only contain lowercase letters, numbers, underscores, or periods.

Google Configuration (required for Google Play export)

Provide either a Product price, or an ID for a Pricing Template created in Google Play.



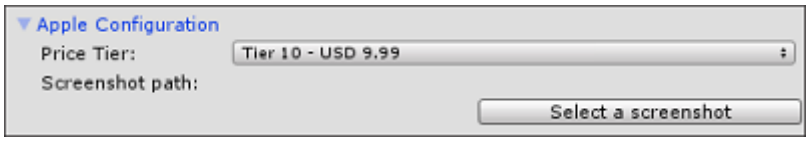
Populating **Google Configuration** fields for Products in the **IAP Catalog** GUI.

Apple Configuration (required for Apple export)

Select a **Pricing Tier** from the dropdown menu. Unity supports predefined Apple price points, but not arbitrary values.

Select a screenshot to upload.

For information on screenshot specs, see Apple's publisher support documentation.



Populating **Apple Configuration** fields for Products in the **IAP Catalog** GUI.

Xiaomi Configuration (required for Xiaomi export)

Select a pricing tier (Chinese yuan) from the drop-down menu. Unity supports predefined Xiaomi price points, but not arbitrary values. For more information on publishing IAP Catalogs for Xiaomi, see documentation on **Xiaomi integration** .

Defining Products in scripts

You can also declare your Product list programmatically using the Purchasing Configuration Builder. You must provide a unique cross-store **Product ID** and **Product Type** for each Product:

```
using UnityEngine;
using UnityEngine.Purchasing;

public class MyIAPManager {
    public MyIAPManager () {
        var builder = ConfigurationBuilder.Instance(StandardPurchasingModule.Ins
        builder.AddProduct("100_gold_coins", ProductType.Consumable);
        // Initialize Unity IAP...
    }
}
```

2018–05–30 Page amended with editorial review

Subscription Product support

Leave feedback

Unity IAP supports Product subscription information queries through the `SubscriptionManager` class. For example code, please review the *IAPDemo.cs* script included in the **Unity IAP** SDK 1.19+.

SubscriptionManager class methods

This class supports the Apple store and Google Play store. For Google Play, this class only supports Products purchased using IAP SDK 1.19+.

Method	Description
`public SubscriptionInfo getSubscriptionInfo()`	Returns a `SubscriptionInfo` object (see below)

SubscriptionInfo class methods

The `SubscriptionInfo` class is a container for a Product's subscription-related information.

Method	Description
`public string getProductId()`	Returns a Product's store ID.
`public DateTime getPurchaseDate()`	Returns the Product's purchase date. For Apple, the purchase date is the date when the subscription was either purchased or renewed. For Google, the purchase date is the date when the subscription was originally purchased.
`public Result isSubscribed()`	Returns a `Result` enum to indicate whether this Product is currently subscribed or not. Non-renewable Products in the Apple store return a `Result.Unsupported` value. Auto-renewable Products in the Apple store and subscription products in the Google Play store return a `Result.True` or `Result.False` value.
`public Result isExpired()`	Returns a Result enum to indicate whether this Product has expired or not. * Non-renewable Products in the Apple store return a `Result.Unsupported` value. * Auto-renewable Products in the Apple store and subscription products in the Google Play store return a `Result.True` or `Result.False` value.

Method	Description
`public Result isCancelled()`	Returns a `Result` enum to indicate whether this Product has been cancelled. A cancelled subscription means the Product is currently subscribed, but will not renew on the next billing date. Non-renewable Products in the Apple store return a `Result.Unsupported` value. Auto-renewable Products in the Apple store and subscription products in the Google Play store return a `Result.True` or `Result.False` value.
`public Result isFreeTrial()`	Returns a `Result` enum to indicate whether this Product is a free trial. * Products in the Google Play store return Result.Unsupported if the application does not support version 6+ of the Android in-app billing API. Non-renewable Products in the Apple store return a `Result.Unsupported` value. Auto-renewable Products in the Apple store and subscription products in the Google Play store return a `Result.True` or `Result.False` value.
`public Result isAutoRenewing()`	Returns a `Result` enum to indicate whether this Product is auto-renewable. Non-renewable Products in the Apple store return a `Result.Unsupported` value. Auto-renewable Products in the Apple store and subscription products in the Google Play store return a `Result.True` or `Result.False` value.
`public TimeSpan getRemainingTime()`	Returns a `TimeSpan` to indicate how much time remains until the next billing date. Products in the Google Play store return `TimeSpan.MaxValue` if the application does not support version 6+ of the Android in-app billing API.
`public Result isIntroductoryPricePeriod()`	Returns a `Result` enum to indicate whether this Product is within an introductory price period. On-renewable Products in the Apple store return a `Result.Unsupported` value. Auto-renewable Products in the Apple store and subscription products in the Google Play store return a `Result.True` or `Result.False` value. Products in the Google Play store return Result. Unsupported if the application does not support version 6+ of the Android in-app billing API.

Method	Description
`public TimeSpan getIntroductoryPricePeriod()`	Returns a `TimeSpan` to indicate how much time remains for the introductory price period. Subscription products with no introductory price period return `TimeSpan.Zero`. Products in the Apple store return TimeSpan.Zero if the application does not support iOS version 11.2+, macOS 10.13.2+, or tvOS 11.2+.
`public long getIntroductoryPricePeriodCycles()`	Returns the number of introductory price periods that can be applied to this Product. Products in the Apple store return 0 if the application does not support iOS version 11.2+, macOS 10.13.2+, or tvOS 11.2+.
`public string getIntroductoryPrice()`	Returns a string to indicate the introductory price of the Product. Products with no introductory price return a `"not available"` value. Apple store Products with an introductory price return a value formatted as `"0.99USD"`. Google Play Products with an introductory price return a value formatted as `"$0.99"`. Products in the Apple store return `"not available"` if the application does not support iOS version 11.2+, macOS 10.13.2+, or tvOS 11.2+.
`public DateTime getExpireDate()`	Returns the date of the Product's next auto-renew or expiration (for a cancelled auto-renewing subscription). Products in the Google Play store return TimeSpan.MaxValue if the application does not support version 6+ of the Android in-app billing API.

2018–05–30 Page published with editorial review

Added Subscription Product support in 2018.1

Initialization

Leave feedback

You must provide an implementation of the `IStoreListener` interface which **Unity IAP** uses to inform your application of purchase-related events.

Call the `UnityPurchasing.Initialize` method to start the initialization process, supplying your listener implementation and configuration.

Note that initialization will not fail if the network is unavailable; Unity IAP will continue attempting to initialize in the background. Initialization will only fail if Unity IAP encounters an unrecoverable problem such as a misconfiguration or IAP being disabled in device settings.

Consequently Unity IAP may take an arbitrary period of time to initialize; indefinitely if the user is in airplane mode. You should design your store accordingly by preventing users from attempting to make purchases if initialization has not completed successfully.

```
using UnityEngine;
using UnityEngine.Purchasing;

public class MyIAPManager : IStoreListener {

    private IStoreController controller;
    private IExtensionProvider extensions;

    public MyIAPManager () {
        var builder = ConfigurationBuilder.Instance(StandardPurchasingModule.Ins
        builder.AddProduct("100_gold_coins", ProductType.Consumable, new IDs
        {
            {"100_gold_coins_google", GooglePlay.Name},
            {"100_gold_coins_mac", MacAppStore.Name}
        });

        UnityPurchasing.Initialize (this, builder);
    }

    /// <summary>
    /// Called when Unity IAP is ready to make purchases.
    /// </summary>
    public void OnInitialized (IStoreController controller, IExtensionProvider e
    {
        this.controller = controller;
        this.extensions = extensions;
    }

    /// <summary>
    /// Called when Unity IAP encounters an unrecoverable initialization error.
```

```
    ///
    /// Note that this will not be called if Internet is unavailable; Unity IAP
    /// will attempt initialization until it becomes available.
    /// </summary>
    public void OnInitializeFailed (InitializationFailureReason error)
    {
    }

    /// <summary>
    /// Called when a purchase completes.
    ///
    /// May be called at any time after OnInitialized().
    /// </summary>
    public PurchaseProcessingResult ProcessPurchase (PurchaseEventArgs e)
    {
        return PurchaseProcessingResult.Complete;
    }

    /// <summary>
    /// Called when a purchase fails.
    /// </summary>
    public void OnPurchaseFailed (Product i, PurchaseFailureReason p)
    {
    }
}
```

Browsing Product Metadata

Leave feedback

Unity IAP retrieves localized product metadata during the initialization process, which you can access via the `IStoreController`'s products field.

```
foreach (var product in controller.products.all) {
    Debug.Log (product.metadata.localizedTitle);
    Debug.Log (product.metadata.localizedDescription);
    Debug.Log (product.metadata.localizedPriceString);
}
```

Initiating Purchases

Leave feedback

When the user wants to buy a product call the `InitiatePurchase` method of the `IStoreController`, identifying the product the user wants to buy.

```
// Example method called when the user presses a 'buy' button
// to start the purchase process.
public void OnPurchaseClicked(string productId) {
    controller.InitiatePurchase(productId);
}
```

Your application will be notified asynchronously of the result, either with an invocation of `ProcessPurchase` for successful purchases or `OnPurchaseFailed` for failures.

Processing Purchases

Leave feedback

The `ProcessPurchase` function of your store listener is called when a purchase completes. Your application should fulfil whatever the user has bought; for example, unlocking local content or sending purchase receipts to a server to update a server-side game model.

A result is returned to indicate whether or not your Application has finished processing the purchase:

Result	Description
PurchaseProcessingResult.Complete	The application has finished processing the purchase and should not be informed of it again.
PurchaseProcessingResult.Pending	The application is still processing the purchase and ProcessPurchase will be called again the next time the Application starts, unless the `ConfirmPendingPurchase` function of `IStoreController` is called.

Note that ProcessPurchase may be called at any point following a successful initialization. If your application crashes during execution of the `ProcessPurchase` handler, then it is invoked again the next time **Unity IAP** initializes, so you may wish to implement your own additional de-duplication.

Reliability

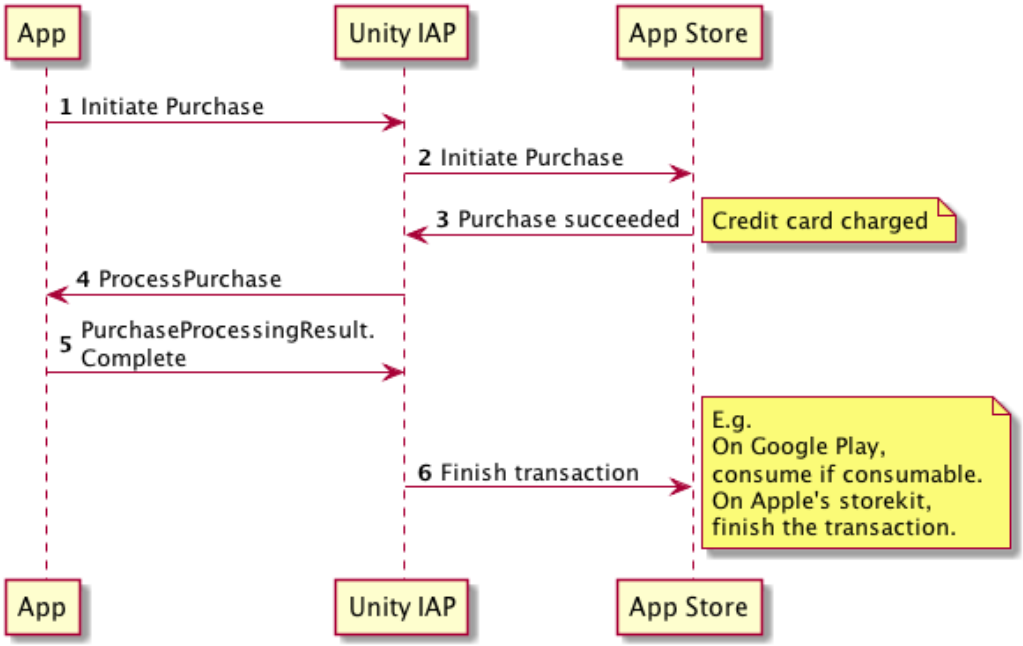
Unity IAP requires explicit acknowledgement of purchases to ensure that purchases are reliably fulfilled in the event of network outages or application crashes. Any purchases that complete while the application is offline will be sent to the application on next initialization.

Completing purchases immediately

When `PurchaseProcessingResult.Complete` is returned, Unity IAP finishes the transaction immediately (as shown in the diagram below).

You **must not** return `PurchaseProcessingResult.Complete` if you are selling consumable products and fulfilling them from a server (for example, providing currency in an online game).

If you do, there is a risk that consumable purchases will be lost if your Application is uninstalled before the cloud save takes place.

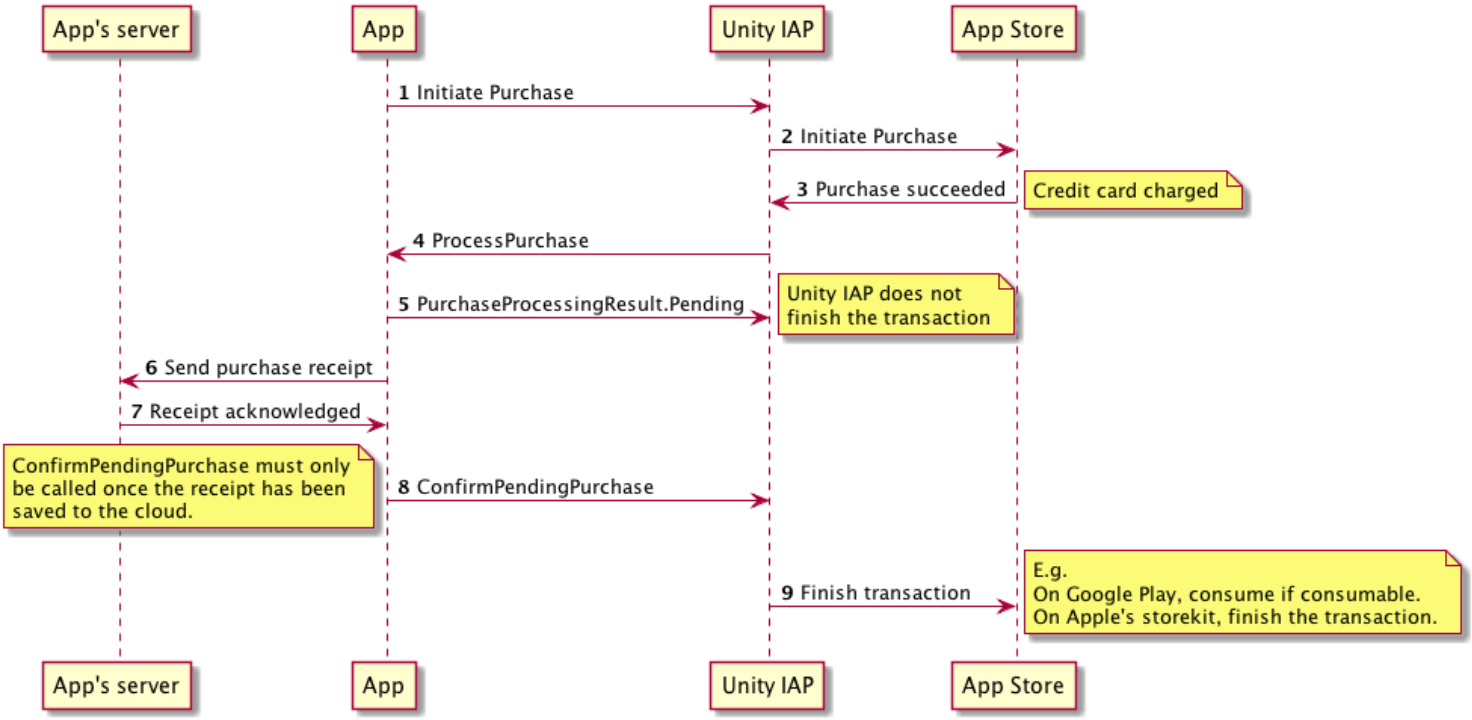


Completing immediately

Saving purchases to the cloud

If you are saving consumable purchases to the cloud, you **must** return `PurchaseProcessingResult.Pending` and call `ConfirmPendingPurchase` only when you have successfully persisted the purchase.

When returning `Pending`, Unity IAP keeps transactions open on the underlying store until confirmed as processed, ensuring consumable purchases are not lost even if a user reinstalls your application while a consumable is in this state.



Pending Purchases

Handling purchase failures

Leave feedback

Purchases may fail for a number of reasons, including network failure, payment failure or device settings. You may wish to check the reason for a purchase failure and prompt the user to take action, though note that not all stores provide fine-grained failure information.

```
/// <summary>
/// Called when a purchase fails.
/// </summary>
public void OnPurchaseFailed (Product i, PurchaseFailureReason p)
{
    if (p == PurchaseFailureReason.PurchasingUnavailable) {
        // IAP may be disabled in device settings.
    }
}
```

Restoring Transactions

Leave feedback

When a user reinstalls your application they should be granted any Non-Consumable or renewable Subscription products they already own. App stores maintain a permanent record of each user's Non-Consumable and renewable Subscription products which **Unity IAP** can retrieve. Non-renewing subscriptions on Apple platforms cannot be restored. If you use non-renewing subscription products on Apple platforms, it is up to you to keep a record of the active subscriptions and sync the subscription between devices.

On platforms that support it (e.g. Google Play and Universal Windows Applications) Unity IAP automatically restores any products the user owns during the first initialization following reinstallation; the `ProcessPurchase` method of your `IStoreListener` will be called for each owned item.

On Apple platforms users must enter their password to retrieve previous transactions so your application must provide users with a button letting them do so. During this process the `ProcessPurchase` method of your `IStoreListener` will be invoked for any items the user already owns.

```
/// <summary>
/// Your IStoreListener implementation of OnInitialized.
/// </summary>
public void OnInitialized(IStoreController controller, IExtensionProvider extens
{
    extensions.GetExtension<IAppleExtensions> ().RestoreTransactions (result =>
        if (result) {
            // This does not mean anything was restored,
            // merely that the restoration process succeeded.
        } else {
            // Restoration failed.
        }
    });
}
```

Purchase Receipts

Leave feedback

Unity IAP provides purchase receipts as a JSON hash containing the following keys and values:

Key	Value
Store	The name of the store in use, such as **GooglePlay** or **AppleAppStore**
TransactionID	This transaction's unique identifier, provided by the store
Payload	Varies by platform, details below.

iOS

Payload varies depending upon the device's **iOS** version.

iOS version	Payload
iOS >= 7	payload is a base 64 encoded App Receipt.
iOS < 7	payload is a SKPaymentTransaction transactionReceipt.

Mac App Store

Payload is a base 64 encoded App Receipt.

Google Play

Payload is a JSON hash with the following keys and values:

Key	Value
json	A JSON encoded string provided by Google; INAPP_PURCHASE_DATA
signature	A signature for the json parameter, as provided by Google; INAPP_DATA_SIGNATURE

Universal Windows Platform

Payload is an XML string as specified by Microsoft

• 2017–05–16 Page amended with no editorial review

Receipt validation

Leave feedback

Receipt validation helps you prevent users from accessing content they have not purchased.

Point of validation

It is best practice to validate the receipt at the point where your application's content is distributed.

Local validation: For client-side content, where all content is contained in the application and is enabled once purchased, the validation should take place on the target device, without the need to connect to a remote server. Unity IAP is designed to support local validation within your application. See **Local validation** below for more information.
Remote validation: For server-side content, where content is downloaded once purchased, the validation should take place on the server before the content is released. Unity does not offer support for server-side validation; however, third-party solutions are available, such as Nobuyori Takahashi's IAP project.

Local validation

If the content that the user is purchasing already exists on the device, the application simply needs to make a decision about whether to unlock it.

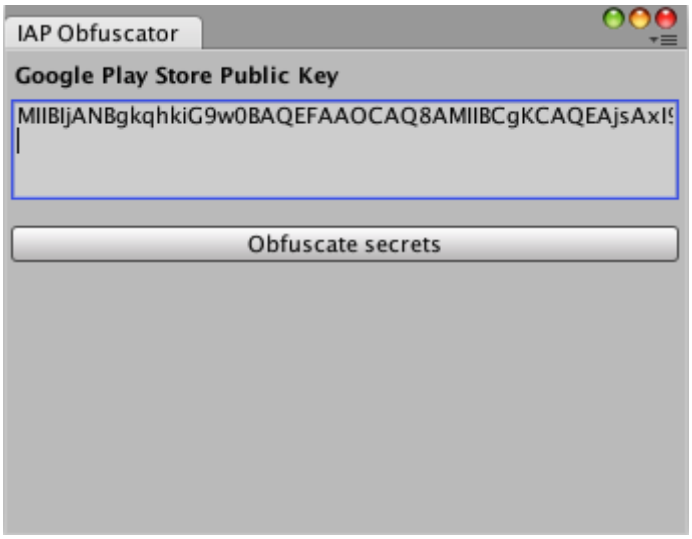
Unity IAP provides tools to help you hide content and to validate and parse receipts through Google Play and Apple stores.

Obfuscating encryption keys

Receipt validation is performed using known encryption keys. For your application, this is an encrypted Google Play public key, and/or Apple's root certificate.

If a user can replace these, they can defeat your receipt validation checks, so it is important to make it difficult for a user to easily find and modify these keys.

Unity IAP provides a tool that can help you obfuscate your encryption keys within your Application. This confuses or jumbles the keys so that it is much harder for a user to acces them. In the Unity menu bar, go to **Window** > **Unity IAP** > **IAP Receipt Validation Obfuscator**.



The Obfuscator window

This window encodes both Apple's root certificate (which is bundled with Unity IAP) and your Google Play public key (from the application's Google Play Developer Console's Services & APIs page) into two different C# files: **AppleTangle** and **GooglePlayTangle**. These are added to your project for use in the next section.

Note that you do not have to provide a Google Play public key if you are only targeting Apple's stores, and vice versa.

Validating receipts

Use the `CrossPlatformValidator` class for validation across both Google Play and Apple stores.

You must supply this class with either your Google Play public key or Apple's root certificate, or both if you wish to validate across both platforms.

The `CrossPlatformValidator` performs two checks:

Receipt authenticity is checked via signature validation.
The application bundle identifier on the receipt is compared to the one in your application. An **InvalidBundleId** exception is thrown if they do not match.

Note that the validator only validates receipts generated on Google Play and Apple platforms. Receipts generated on any other platform, including fakes generated in the Editor, throw an **IAPSecurityException**.

If you try to validate a receipt for a platform that you haven't supplied a secret key for, a **MissingStoreSecretException** is thrown.

```
public PurchaseProcessingResult ProcessPurchase (PurchaseEventArgs e)
{
    bool validPurchase = true; // Presume valid for platforms with no R.V.

    // Unity IAP's validation logic is only included on these platforms.
#if UNITY_ANDROID || UNITY_IOS || UNITY_STANDALONE_OSX
    // Prepare the validator with the secrets we prepared in the Editor
    // obfuscation window.
    var validator = new CrossPlatformValidator(GooglePlayTangle.Data(),
        AppleTangle.Data(), Application.bundleIdentifier);

    try {
        // On Google Play, result has a single product ID.
        // On Apple stores, receipts contain multiple products.
        var result = validator.Validate(e.purchasedProduct.receipt);
        // For informational purposes, we list the receipt(s)
        Debug.Log("Receipt is valid. Contents:");
        foreach (IPurchaseReceipt productReceipt in result) {
            Debug.Log(productReceipt.productID);
            Debug.Log(productReceipt.purchaseDate);
            Debug.Log(productReceipt.transactionID);
```

```
        }
    } catch (IAPSecurityException) {
        Debug.Log("Invalid receipt, not unlocking content");
        validPurchase = false;
    }
#endif

    if (validPurchase) {
        // Unlock the appropriate content here.
    }

    return PurchaseProcessingResult.Complete;
}
```

It is important you check not just that the receipt is valid, but also what information it contains. A common technique by users attempting to access content without purchase is to supply receipts from other products or applications. These receipts are genuine and do pass validation, so you should make decisions based on the product IDs parsed by the **CrossPlatformValidator**.

Store-specific details

Different stores have different fields in their purchase receipts. To access store-specific fields, `IPurchaseReceipt` can be downcast to two different subtypes: `GooglePlayReceipt` and `AppleInAppPurchaseReceipt`.

```
var result = validator.Validate(e.purchasedProduct.receipt);
Debug.Log("Receipt is valid. Contents:");
foreach (IPurchaseReceipt productReceipt in result) {
    Debug.Log(productReceipt.productID);
    Debug.Log(productReceipt.purchaseDate);
    Debug.Log(productReceipt.transactionID);

    GooglePlayReceipt google = productReceipt as GooglePlayReceipt;
    if (null != google) {
        // This is Google's Order ID.
        // Note that it is null when testing in the sandbox
        // because Google's sandbox does not provide Order IDs.
        Debug.Log(google.transactionID);
        Debug.Log(google.purchaseState);
        Debug.Log(google.purchaseToken);
    }

    AppleInAppPurchaseReceipt apple = productReceipt as AppleInAppPurchaseReceip
```

```
    if (null != apple) {
        Debug.Log(apple.originalTransactionIdentifier);
        Debug.Log(apple.subscriptionExpirationDate);
        Debug.Log(apple.cancellationDate);
        Debug.Log(apple.quantity);
    }
}
```

Parsing raw Apple receipts

Use the `AppleValidator` class to extract detailed information about an Apple receipt. Note that this class only works with **iOS** App receipts from version 7.0 onwards, not Apple's deprecated transaction receipts.

```
#if UNITY_ANDROID || UNITY_IOS || UNITY_STANDALONE_OSX
var builder = ConfigurationBuilder.Instance(StandardPurchasingModule.Instance())
// Get a reference to IAppleConfiguration during IAP initialization.
var appleConfig = builder.Configure<IAppleConfiguration>();
var receiptData = System.Convert.FromBase64String(appleConfig.appReceipt);
AppleReceipt receipt = new AppleValidator(AppleTangle.Data()).Validate(receiptDa

Debug.Log(receipt.bundleID);
Debug.Log(receipt.receiptCreationDate);
foreach (AppleInAppPurchaseReceipt productReceipt in receipt.inAppPurchaseReceip
    Debug.Log(productReceipt.transactionIdentifier);
    Debug.Log(productReceipt.productIdentifier);
}
#endif
```

The `AppleReceipt` type models Apple's ASN1 receipt format. See Apple's documentation for an explanation of its fields.

Store Extensions

Leave feedback

Stores may offer unique functionality that does not fit into the normal cross-platform purchase flow. This extended functionality is accessed via the `IExtensionProvider` which is provided to your application when **Unity IAP** initializes successfully.

It is not necessary to use platform-dependent compilation when using extensions; each extension comes with a fake no-op implementation which is used when running on a platform that does not offer the extended functionality.

For example, the following snippet accesses the `RefreshReceipt` mechanism Apple offers to fetch a refreshed App Receipt from Apple's servers. It can be compiled on any Unity IAP platform, and if you were to run it on a non Apple platform such as Android it would have no effect; the supplied lambda would never be invoked.

```
/// <summary>
/// Called when Unity IAP is ready to make purchases.
/// </summary>
public void OnInitialized (IStoreController controller, IExtensionProvider exten
{
    extensions.GetExtension<IAppleExtensions> ().RefreshAppReceipt (result => {
        if (result) {
            // Refresh finished successfully.
        } else {
            // Refresh failed.
        }
    });
}
```

Cross-store installation issues with Android in-app purchase stores

Leave feedback

There are cross-store installation issues when using shared Android bundle identifiers to publish to multiple Android in-app purchase stores (such as Samsung and Google) simultaneously. This page describes the results of these conflicts, and how to resolve the issue.

The cross-store install scenario is one where a user installs an application from one store (store A), and then upgrades the installation with an application from another store (store B). The opportunity to upgrade a user's installation belongs to the store hosting the newest application version. This scenario can arise when both builds of the application use the same Android bundle identifier and signing key.

For example, `com.foo.bar` is published to Google Play and Amazon Appstore. A user who has both App stores installed could install `com.foo.bar` from Amazon Apps and receive an updated version from Google Play. This could result in them losing IAP digital product transactions, and being unable to restore previously made transactions.

Impact of cross-store installation issues

Cross-store conflict can cause intractable end-user problems with applications using IAP. Users may find they lose purchases upon uninstallation/reinstallation and lose any in-flight purchases interrupted by a cross-store upgrade.

The resulting IAP problems are:

Losing incomplete purchases. This occurs if the user upgrades after a purchase has been approved but before the application has acknowledged receiving the purchase from the store's billing system. This can arise when an application fails to synchronize with a cloud inventory service in a timely fashion, or the app crashes before saving this new IAP to the local inventory database.

Withholding purchases from store A. This occurs if a user reinstalls an application from store B after completing IAP transactions on store A. They will not be able to restore those transactions. Many stores explicitly disallow applications from accessing other stores' IAP systems. To workaround this utilize a cloud inventory system along with its dependent user identity service.

Divergent IAP product lists. This occurs if the upgraded product list in store B's application diverges from the list in store A's version. You may provide divergent IAP cataloges. This may result in an application error when an inconsistent local inventory is read by the store B version, or the user may lose the inventory they previously purchased which is unavailable through the store B version.

Resolving cross-store installation issues

There are two ways to resolve cross-store conflicts:

use unique signing keys for each store (This results in upgrade error messages from the conflicting store), or
use unique bundle identifiers for each store (This may result in duplicate installed apps for the user).

iOS & Mac App Stores

Leave feedback

Extended functionality

Reading the App Receipt

An App Receipt is stored on the device's local storage and can be read as follows:

```
var builder = ConfigurationBuilder.Instance(StandardPurchasingModule.Instance())
string receipt = builder.Configure<IAppleConfiguration>().appReceipt;
```

Checking for payment restrictions

In App Purchases may be restricted in a device's settings, which can be checked for as follows:

```
var builder = ConfigurationBuilder.Instance(StandardPurchasingModule.Instance())
bool canMakePayments = builder.Configure<IAppleConfiguration>().canMakePayments;
```

Restoring transactions

On Apple platforms users must enter their password to retrieve previous transactions so your application must provide users with a button letting them do so. During this process the `ProcessPurchase` method of your `IStoreListener` will be invoked for any items the user already owns.

```
/// <summary>
/// Your IStoreListener implementation of OnInitialized.
/// </summary>
public void OnInitialized(IStoreController controller, IExtensionProvider extens
{
    extensions.GetExtension<IAppleExtensions> ().RestoreTransactions (result =>
        if (result) {
            // This does not mean anything was restored,
            // merely that the restoration process succeeded.
        } else {
            // Restoration failed.
        }
    });
}
```

Refreshing the App Receipt

Apple provides a mechanism to fetch a new App Receipt from their servers, typically used when no receipt is currently cached in local storage; SKReceiptRefreshRequest.

Note that this will prompt the user for their password.

Unity IAP makes this method available as follows:

```
/// <summary>
/// Your IStoreListener implementation of OnInitialized.
/// </summary>
public void OnInitialized(IStoreController controller, IExtensionProvider extens
{
    extensions.GetExtension<IAppleExtensions> ().RefreshAppReceipt (receipt => {
        // This handler is invoked if the request is successful.
        // Receipt will be the latest app receipt.
        Console.WriteLine(receipt);
    },
    () => {
        // This handler will be invoked if the request fails,
        // such as if the network is unavailable or the user
        // enters the wrong password.
    });
}
```

Ask to Buy

iOS 8 introduced a new parental control feature called Ask to Buy.

Ask to Buy purchases defer for parent approval. When this occurs, **Unity IAP** sends your app a notification as follows:

```
/// This is called when Unity IAP has finished initialising.
public void OnInitialized(IStoreController controller, IExtensionProvider extens
{
    extensions.GetExtension<IAppleExtensions>().RegisterPurchaseDeferredListener
        Console.WriteLine(product.definition.id);
    });
}
```

Enable “Ask-to-Buy” simulation in the Sandbox App Store

The sample class below demonstrates how to access `IAppleExtensions` in order to enable Ask-to-Buy simulation in the Sandbox App Store:

```
using UnityEngine;
using UnityEngine.Purchasing;

public class AppleSimulateAskToBuy : MonoBehaviour {
    public void SetSimulateAskToBuy(bool shouldSimulateAskToBuy) {
        if (Application.platform == RuntimePlatform.IPhonePlayer) {
            IAppleExtensions extensions = IAPButton.IAPButtonStoreManager.Instan
            extensions.simulateAskToBuy = shouldSimulateAskToBuy;
        }
    }
}
```

When the purchase is approved or rejected, your store's normal `ProcessPurchase` or `OnPurchaseFailed` listener methods are invoked.

Transaction Receipts

Sometimes consumable Ask to Buy purchases don't show up in the App Receipt, in which case you cannot validate them using that receipt. However, **iOS** provides a Transaction Receipt that contains all purchases, including Ask to Buy. Access the most recent Transaction Receipt string for a given `Product` using `IAppleExtensions`.

Note: Transaction Receipts are not available for Mac builds. Requesting a Transaction Receipt on a Mac build results in an empty string.

```
#if UNITY_PURCHASING

using System;
using UnityEngine;
using UnityEngine.Purchasing;

public class AskToBuy : MonoBehaviour, IStoreListener
{
    // Unity IAP objects
    private IStoreController m_Controller;
    private IAppleExtensions m_AppleExtensions;

    public AskToBuy ()
    {
        var builder = ConfigurationBuilder.Instance (StandardPurchasingModule.In
        builder.AddProduct ("100_gold_coins", ProductType.Consumable, new IDs {
```

```
        { "100_gold_coins_google", GooglePlay.Name },
        { "100_gold_coins_mac", MacAppStore.Name }
    });

    UnityPurchasing.Initialize (this, builder);
}

/// <summary>
/// This will be called when Unity IAP has finished initialising.
/// </summary>
public void OnInitialized (IStoreController controller, IExtensionProvider e
{
    m_Controller = controller;
    m_AppleExtensions = extensions.GetExtension<IAppleExtensions> ();

    // On Apple platforms we need to handle deferred purchases caused by App
    // On non-Apple platforms this will have no effect; OnDeferred will neve
    m_AppleExtensions.RegisterPurchaseDeferredListener (OnDeferred);
}

/// <summary>
/// This will be called when a purchase completes.
/// </summary>
public PurchaseProcessingResult ProcessPurchase (PurchaseEventArgs e)
{
    if (Application.platform == RuntimePlatform.IPhonePlayer ||
        Application.platform == RuntimePlatform.tvOS) {
        string transactionReceipt = m_AppleExtensions.GetTransactionReceiptF
        Console.WriteLine (transactionReceipt);
        // Send transaction receipt to server for validation
    }
    return PurchaseProcessingResult.Complete;
}

/// <summary>
/// Called when Unity IAP encounters an unrecoverable initialization error.
///
/// Note that this will not be called if Internet is unavailable; Unity IAP
/// will attempt initialization until it becomes available.
/// </summary>
public void OnInitializeFailed (InitializationFailureReason error)
{
}

/// <summary>
/// Called when a purchase fails.
```

```
    /// </summary>
    public void OnPurchaseFailed (Product i, PurchaseFailureReason p)
    {
    }

    /// <summary>
    /// iOS Specific.
    /// This is called as part of Apple's 'Ask to buy' functionality,
    /// when a purchase is requested by a minor and referred to a parent
    /// for approval.
    ///
    /// When the purchase is approved or rejected, the normal purchase events
    /// will fire.
    /// </summary>
    /// <param name="item">Item.</param>
    private void OnDeferred (Product item)
    {
        Debug.Log ("Purchase deferred: " + item.definition.id);
    }
}

#endif // UNITY_PURCHASING
```

Unlike App Receipts, you cannot validate Transaction Receipts locally. Instead, you must send the receipt string to a remote server for validation. If you already use a remote server to validate App Receipts, send Transaction Receipts to the same Apple endpoint, to receive a JSON response.

Example JSON response:


```
{
    "receipt": {
        "original_purchase_date_pst": "2017-11-15 15:25:20 America/Los_Angeles",
        "purchase_date_ms": "1510788320209",
        "unique_identifier": "0ea7808637555b2c633eb07aa1cb0894c821a6f9",
        "original_transaction_id": "1000000352597239",
        "bvrs": "0",
        "transaction_id": "1000000352597239",
        "quantity": "1",
        "unique_vendor_identifier": "01B57C2E-9E91-42FF-9B0D-4983175D6694",
        "item_id": "1141751870",
        "original_purchase_date": "2017-11-15 23:25:20 Etc/GMT",
        "product_id": "100.gold.coins",
        "purchase_date": "2017-11-15 23:25:20 Etc/GMT",
```

```
        "is_trial_period": "false",
        "purchase_date_pst": "2017-11-15 15:25:20 America/Los_Angeles",
        "bid": "com.unity3d.unityiap.demo",
        "original_purchase_date_ms": "1510788320209"
    },
    "status": 0
}
```

Intercepting Apple promotional purchases

Apple allows you to promote in-game purchases through your app's product page. Unlike conventional in-app purchases, Apple promotional purchases initiate directly from the App Store on iOS and tvOS. The App Store then launches your app to complete the transaction, or prompts the user to download the app if it isn't installed.

The `IAppleConfiguration SetApplePromotionalPurchaseInterceptor` callback method intercepts Apple promotional purchases. Use this callback to present parental gates, send **analytics** events, or perform other functions before sending the purchase to Apple. The callback uses the `Product` that the user attempted to purchase. You must call `IAppleExtensions.ContinuePromotionalPurchases()` to continue with the promotional purchase. This will initiate any queued-up payments.

If you do not set the callback, promotional purchases go through immediately and call `ProcessPurchase` with the result.

Note: Calling these APIs on other platforms has no effect.

```
private IAppleExtensions m_AppleExtensions;

public void Awake() {
    var module = StandardPurchasingModule.Instance();
    var builder = ConfigurationBuilder.Instance(module);
        // On iOS and tvOS we can intercept promotional purchases that come dire
        // the App Store.
        // On other platforms this will have no effect; OnPromotionalPurchase wi
        // called.
    builder.Configure<IAppleConfiguration>().
     SetApplePromotionalPurchaseInterceptorCallback(OnPromotionalPurchase);
    Debug.Log("Setting Apple promotional purchase interceptor callback");
}

public void OnInitialized(IStoreController controller, IExtensionProvider extens
    m_AppleExtensions = extensions.GetExtension<IAppleExtensions>();
    foreach (var item in controller.products.all) {
        if (item.availableToPurchase) {
            // Set all these products to be visible in the user's App Store
```

```
            m_AppleExtensions.SetStorePromotionVisibility(item, AppleStorePromot
        }
    }
}

private void OnPromotionalPurchase(Product item) {
    Debug.Log("Attempted promotional purchase: " + item.definition.id);
    // Promotional purchase has been detected.
    // Handle this event by, e.g. presenting a parental gate.
    // Here, for demonstration purposes only, we will wait five seconds before c
    // the purchase.
    StartCoroutine(ContinuePromotionalPurchases());
}

private IEnumerator ContinuePromotionalPurchases() {
    Debug.Log("Continuing promotional purchases in 5 seconds");
    yield return new WaitForSeconds(5);
    Debug.Log("Continuing promotional purchases now");
    m_AppleExtensions.ContinuePromotionalPurchases (); // iOS and tvOS only
}
```

Testing

To test on Apple stores you must be using an iTunes connect test account, which can be created in iTunes connect.

Sign out of the App Store on the iOS device or laptop, launch your application and you will be prompted to log in when you attempt either a purchase or to restore transactions.

If you receive an initialization failure with a reason of `NoProductsAvailable`, follow this checklist:

iTunes Connect product identifiers must exactly match the product identifiers supplied to Unity IAP
In-App purchases must be enabled for your application in iTunes Connect
Products must be cleared for sale in iTunes Connect
It may take many hours for newly created iTunes Connect products to be available for purchase
You must agree to the latest iTunes Connect developer agreements and have active bank details

Mac App Store

When building a desktop Mac build you must select `Mac App Store validation` within Unity's build settings.

Once you have built your App, you must update its info.plist file with your bundle identifier and version strings. Right click on the `.app` file and click `show package contents`, locate the `info.plist` file and update the `CFBundleIdentifier` string to your application's bundle identifier.

You must then sign, package and install your application. You will need to run the following commands from an OSX terminal:

```
codesign -f --deep -s "3rd Party Mac Developer Application: " your.app/Contents/
codesign -f --deep -s "3rd Party Mac Developer Application: " your.app
productbuild --component your.app /Applications --sign "3rd Party Mac Developer
```

To sign the bundle, you may first need to remove the Contents.meta file if it exists: `your.app/Contents/Plugins/unitypurchasing.bundle/Contents.meta`

In order to install the package correctly you must delete the unpackaged .app file before running the newly created package.

You must then launch your App from the Applications folder. The first time you do so, you will be prompted to enter your iTunes account details, for which you should enter your iTunes Connect test user account login. You will then be able to make test purchases against the sandbox environment.

2018–05–30 Page published with editorial review
Ask to Buy, Transaction Receipts, and Intercepting Receipts added in 2017.3

Universal Windows Platform

Leave feedback

Simulator

Unity IAP features support for Microsoft's **In App Purchase** simulator, which allows you to test IAP purchase flows on devices before publishing your application.

The simulator can be enabled when configuring **Unity IAP** before initialization, as follows:

```
var builder = ConfigurationBuilder.Instance(StandardPurchasingModule.Instance())
builder.Configure<IMicrosoftConfiguration>().useMockBillingSystem = true;
```

Make sure you disable the mock billing system before publishing your application.

• 2017–05–16 Page amended with no editorial review

Google Play

Leave feedback

Consumables

Unity IAP uses V3 of Google's Billing API, which features the concept of consumable products and explicit consumption API calls.

You should create all of your consumable products in the Google Publisher dashboard as 'Managed' products. **Unity IAP** will take care of consuming them when your application confirms a purchase.

Testing

Before publishing your application you must test your in-app purchases on an Android device as an alpha or beta distribution.

Note that whilst your Alpha or Beta **APK** must be published to test your IAPs, this does not mean your App has to be publicly visible in the Google Play store.

In order to perform a complete end-to-end test of your in-app purchases, you must do so whilst signed into a device using a test account.

Please note the following:

You must upload a signed, release version of your APK to Google Play that is published as an alpha or beta distribution.
The version number of the APK that you upload must match the version number of the APK that you test with.
After entering your purchase metadata into the Google Play publisher console, it may take up to 24 hours before you are able to purchase your in-app purchases using a test account.

Amazon Appstore and Amazon Underground Store

Leave feedback

Extended functionality

Amazon User ID

To fetch the current Amazon User ID for other Amazon services, use the `IAmazonExtensions`:

```
public void OnInitialized
    (IStoreController controller, IExtensionProvider extensions)
{
    string amazonUserId =
        extensions.GetExtension<IAmazonExtensions>().amazonUserId;
    // ...
}
```

Sandbox testing in Amazon

To use Amazon's local Sandbox testing app, generate a JSON description of your product catalog on the device's SD card using the `IAmazonConfiguration` extended configuration:

```
var builder = ConfigurationBuilder.Instance(
StandardPurchasingModule.Instance());
// Define your products.
builder.AddProduct("someConsumable", ProductType.Consumable);
// Write a product description to the SD card
// in the appropriate location.
builder.Configure<IAmazonConfiguration>()
    .WriteSandboxJSON(builder.products);
```

When using this method to write product descriptions to the SD card, declare the Android permission to write to external storage in the test app's manifest:

```
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
```

Remove this extra permission before publishing, if appropriate.

Amazon Sandbox is now set up for local testing. For more information, please see Amazon's App Tester documentation.

Samsung Galaxy apps

Leave feedback

Extended functionality

Developer mode testing

Developer mode allows you to carry out IAP testing without incurring real-world monetary charges for products. To get started, create your configuration with an `ISamsungAppsConfiguration` instance, with its mode set to `SamsungAppsMode.AlwaysSucceed`:

```
var builder = ConfigurationBuilder.Instance(StandardPurchasingModule.Instance())

// Enable "developer mode" for purchases, not requiring real-world money

// SamsungAppsMode has: Production (developer mode "off"), AlwaysSucceed, Always

builder.Configure<ISamsungAppsConfiguration>().SetMode(SamsungAppsMode.AlwaysSuc
```

Restoring transactions

Users restore transactions to maintain access to content they've already purchased (for example, when they upgrade to a new phone, they don't lose all of the items they purchased on the old phone). The Samsung Galaxy App Store does not require previous transactions to be restored. However, you could improve in-app usability by providing users with a button allowing them to restore their purchases, for instance if they have installed the app on a different device.

During this process, the `ProcessPurchase` function of your `IStoreListener` is invoked for any items the user already owns. The following example illustrates such a call. This could be called from a **Restore Purchases** button:

```
/// <summary>

/// Your IStoreListener implementation of OnInitialized.

/// </summary>

public void OnInitialized(IStoreController controller, IExtensionProvider extens

{

    // The ProcessPurchase function is invoked for any items the user already ow

    extensions.GetExtension<ISamsungAppsExtensions>().RestoreTransactions(result
```

```
        if (result) {

            // This does not mean anything has been restored,

            // just that the restoration process succeeded.

        } else {

            // Restoration failed.

        }

    });

}
```

On Samsung Galaxy platforms, users may be required to input their Samsung Galaxy App Store password to retrieve previous transactions, if they haven’t already done so.

CloudMoolah MOO store

Leave feedback

The CloudMoolah MOO Store is a commercial service that enables people to buy and sell mobile apps. It also provides In-App Purchasing (IAP) services for apps distributed through the store. This guide describes the end-to-end process for a developer adding IAP services to a game that they publish to the CloudMoolah MOO Store.

The MOO Store serves Asian markets including the Philippines, Malaysia, Vietnam, Singapore, and Thailand. Users maintain a digital wallet, which they can fund using a variety of payment providers, such as mobile-pay, banks, and convenience store prepaid cards.

Although compatible with Unity, the CloudMoolah MOO Store is not a Unity product. For more information, see the CloudMoolah Developer Portal website at dev.cloudmoolah.com.

The general workflow for publishing your game to the store and adding IAP to your game is as follows:

Register your game as an app in the CloudMoolah store
Add items in the store that you can add to your game for purchase
Add the CloudMoolah store initialization code to your game
Test the integration
Publish the game to the CloudMoolah store

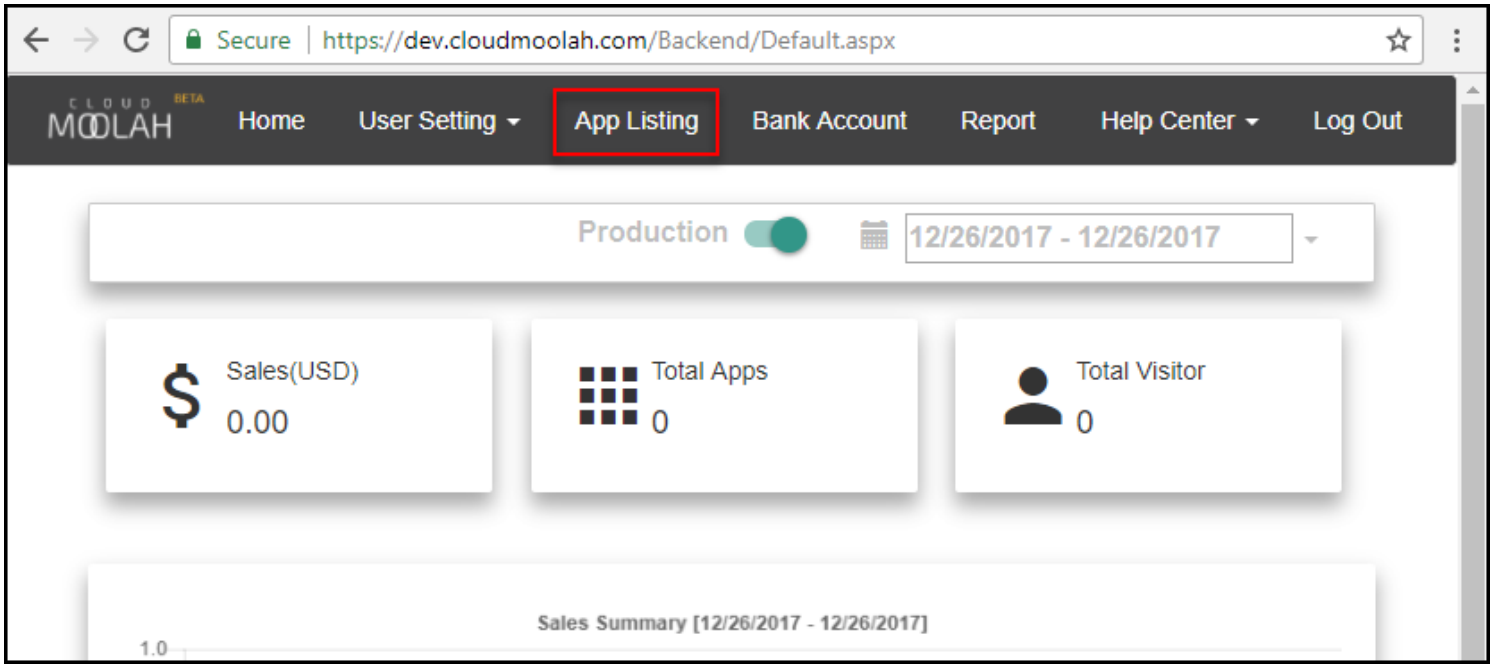
Getting started

Your game must be built as an Android app. For information on building as an Android app, see Getting started with Android development.

To register your game as an available app in the CloudMoolah store:

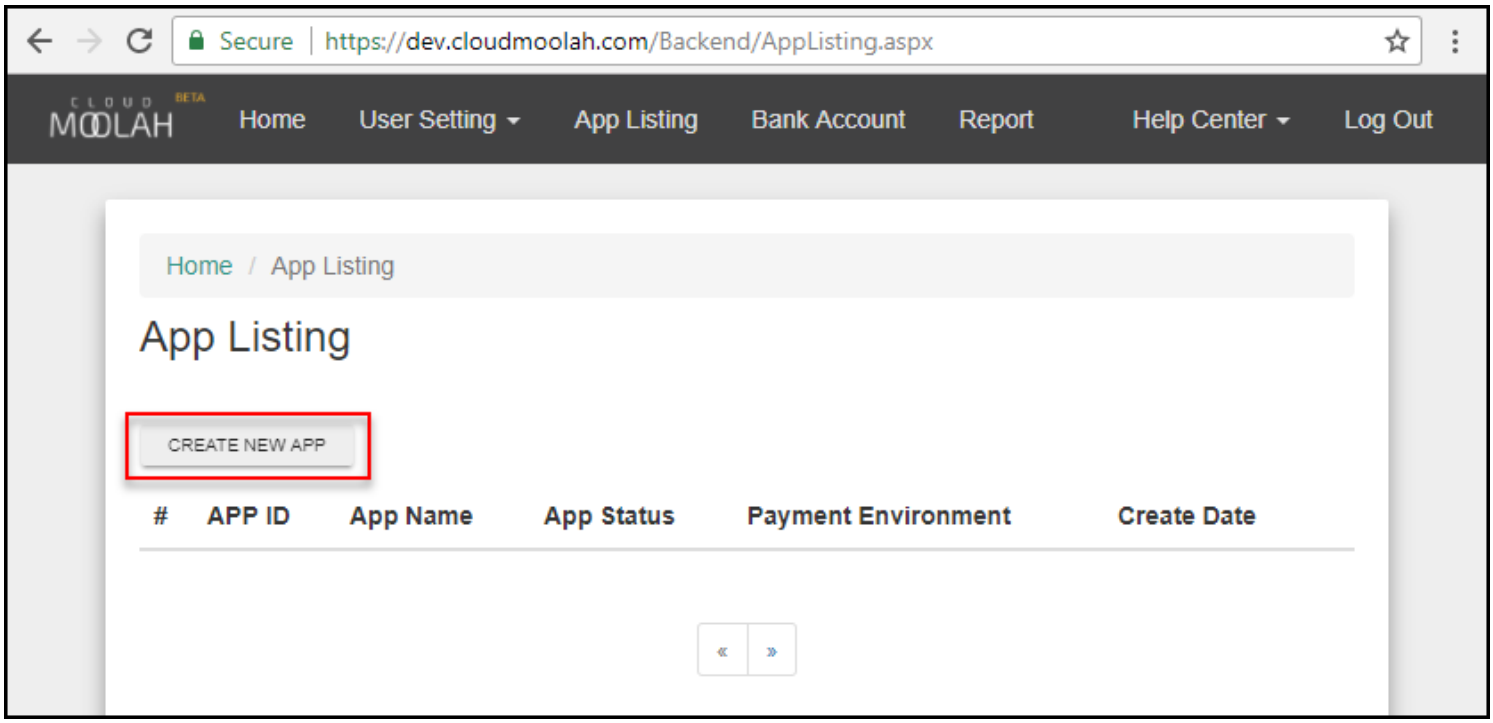
Sign in to the CloudMoolah Developer Portal.

Click **App Listing**.



CloudMoolah Store dashboard menu

On the App Listing page, click the **CREATE NEW APP** button.



CloudMoolah create new app button

On the **App Detail** page, enter the following information:

In **App Name**, enter a name that will identify your game in the CloudMoolah store.

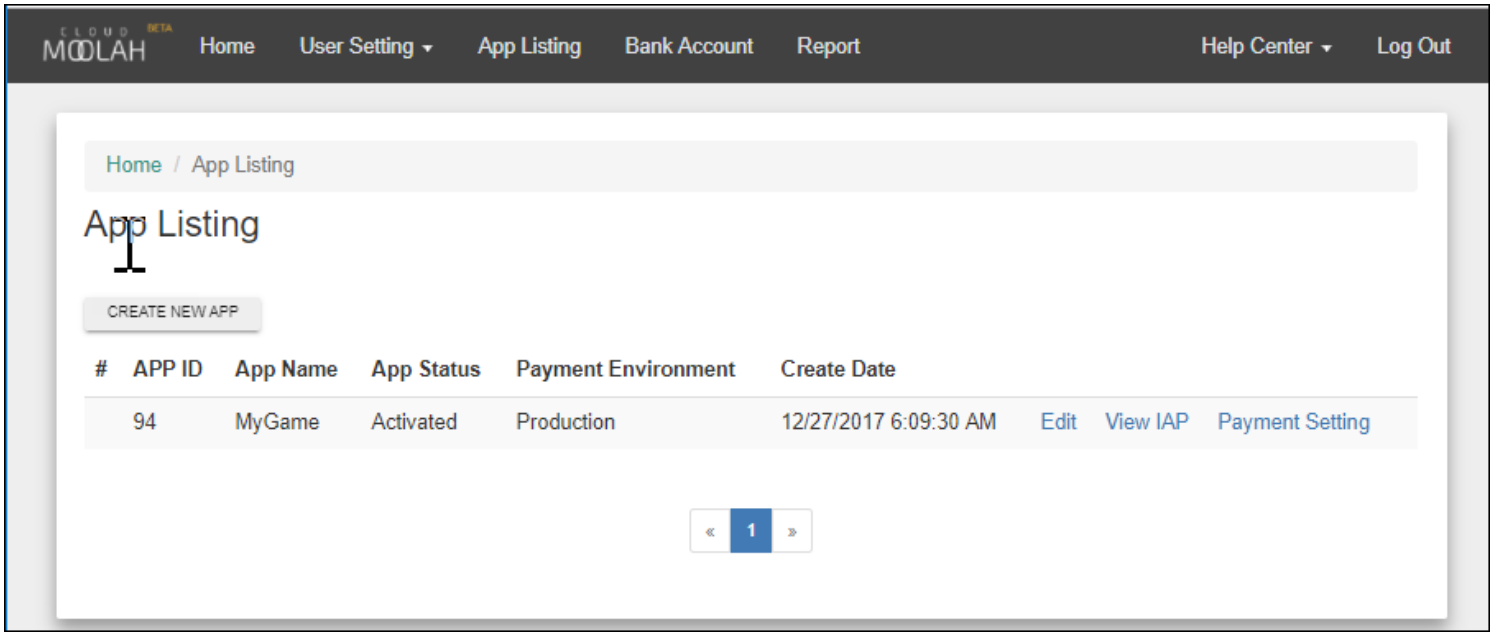
In **BundelID**, enter the Package name you used for your game in the Unity Editor.

In **Currency Name**, enter a name for the currency that players will use to complete the in-app purchase.

To create a HashKey that you will use in the initialization code for your app, click **GENERATE** next to the **HMAC/HashKey** field.

In the **Notification HMAC** field, enter the App ID. This value is used during initialization.

Click **Save**.



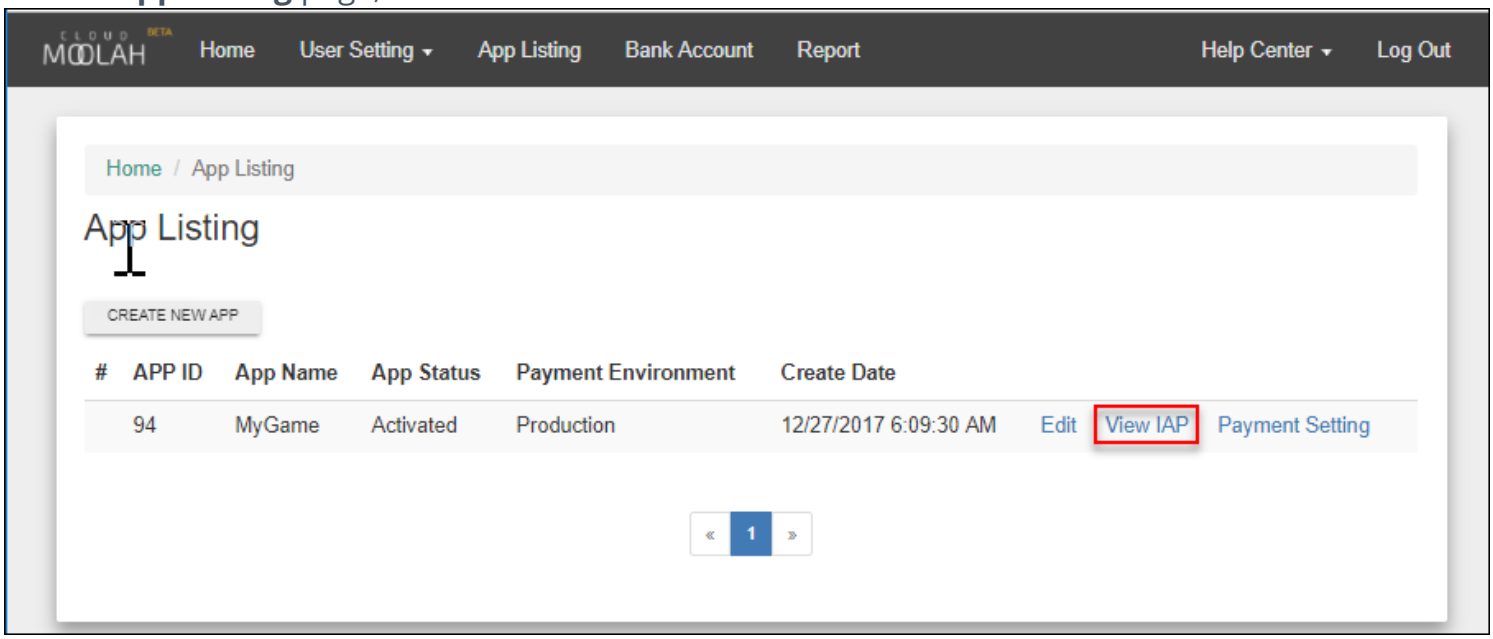
CloudMoolah create new app result

Add items that your users can purchase

An in-app purchase is the transaction of exchanging money for digital goods inside an application. A platform's store allows users to purchase products, which are digital goods. These products have an identifier, which is typically a string data type.

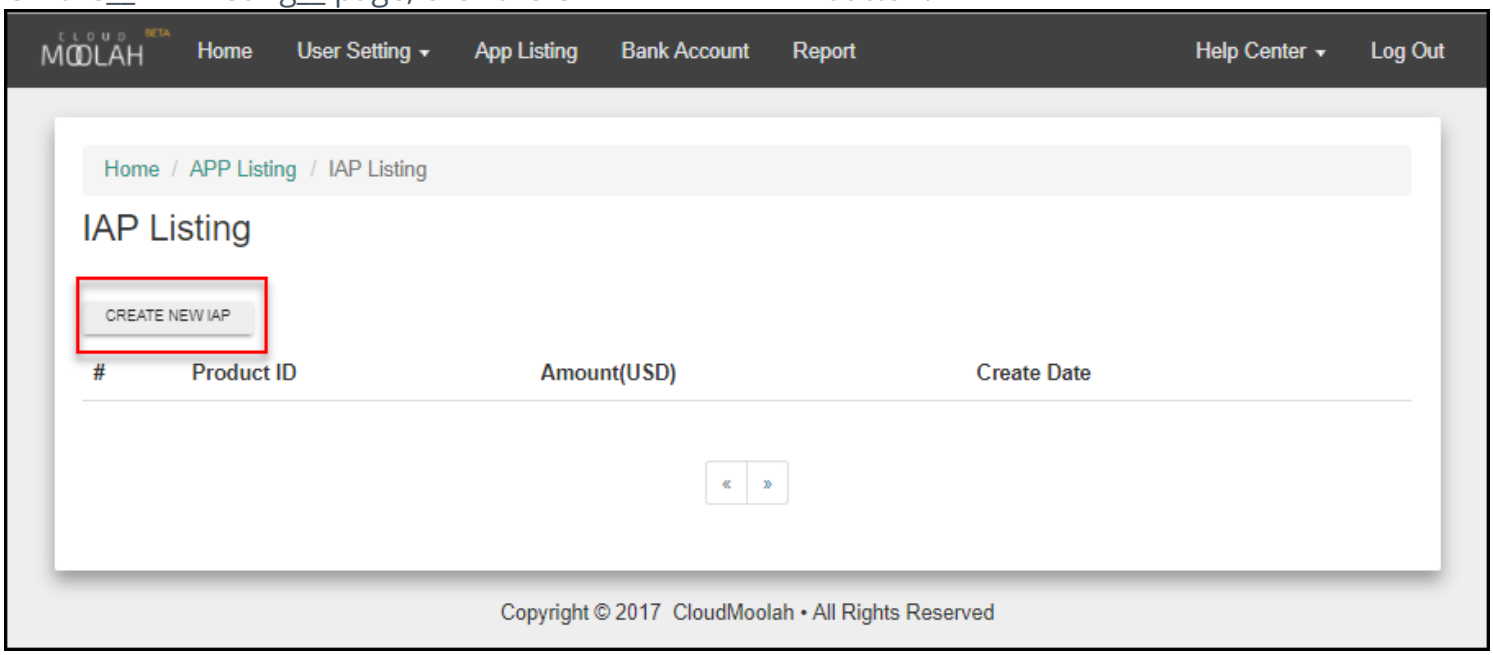
To add a purchasable item in the CloudMoolah Developer portal:

On the **App Listing** page, click **View IAP**.



CloudMoolah view IAP button

On the__ IAP Listing__ page, click the **CREATE NEW IAP** button.



CloudMoolah create new IAP button

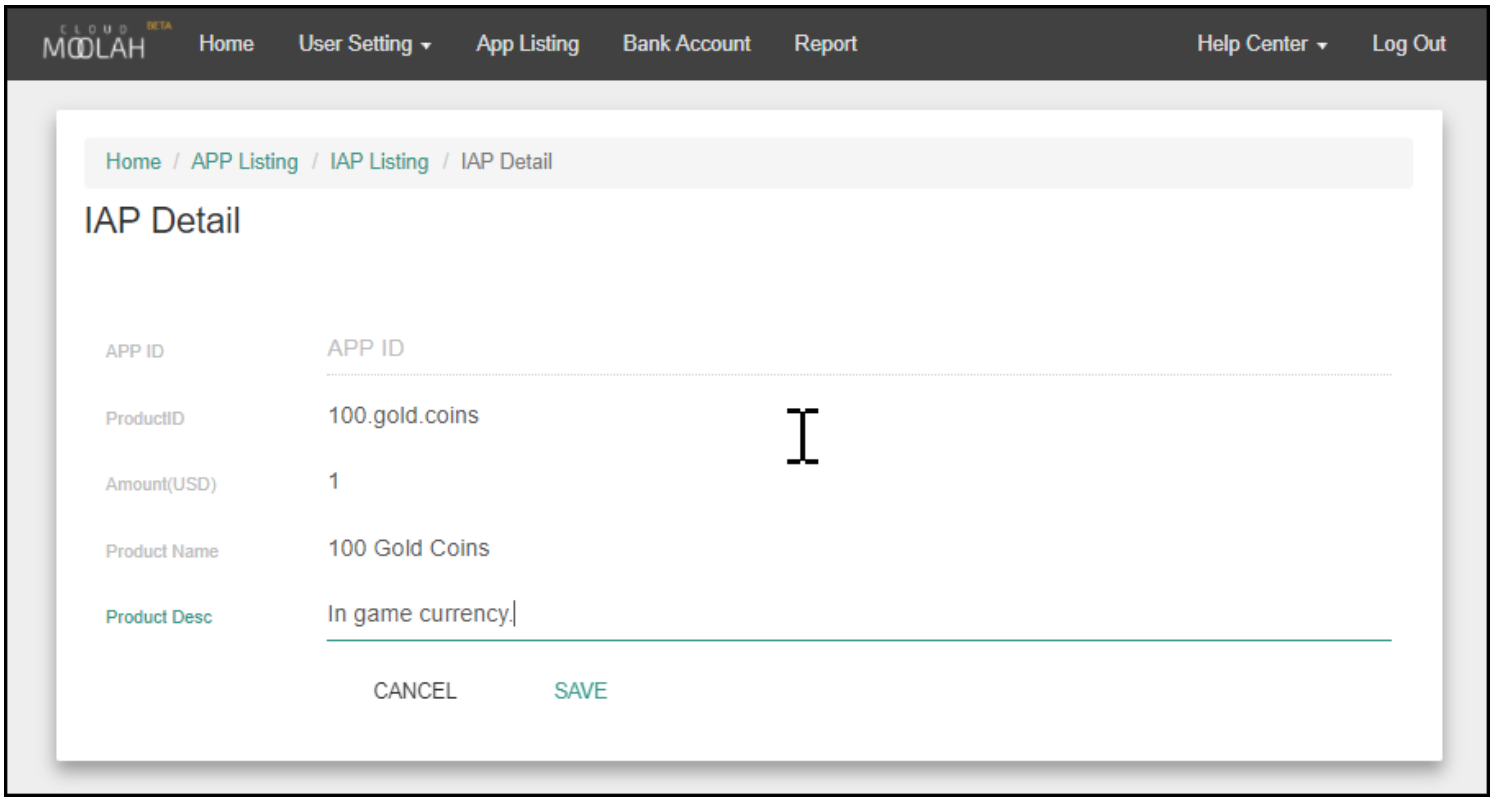
On the **IAP Detail** page, enter the following information:

In the **ProductID** field, enter a string identifier that you will use in your code to enable purchase of the item in your game.

In the **Amount** field, enter the number cost of the item in the default currency you specified when you registered your game.

In the **Product Name** field, enter a name for the product.

In the **Product Description**, enter a description for the product.



CloudMoolah create new IAP result

Click **SAVE**.

The item now appears on the **IAP Listing** page and you can reference it from your game to offer it to your users.

Add IAP to your game

To configure IAP in your game:

Retrieve the appKey and hashKey for your app's product from the MOO Store Developer Portal and store them for later use.

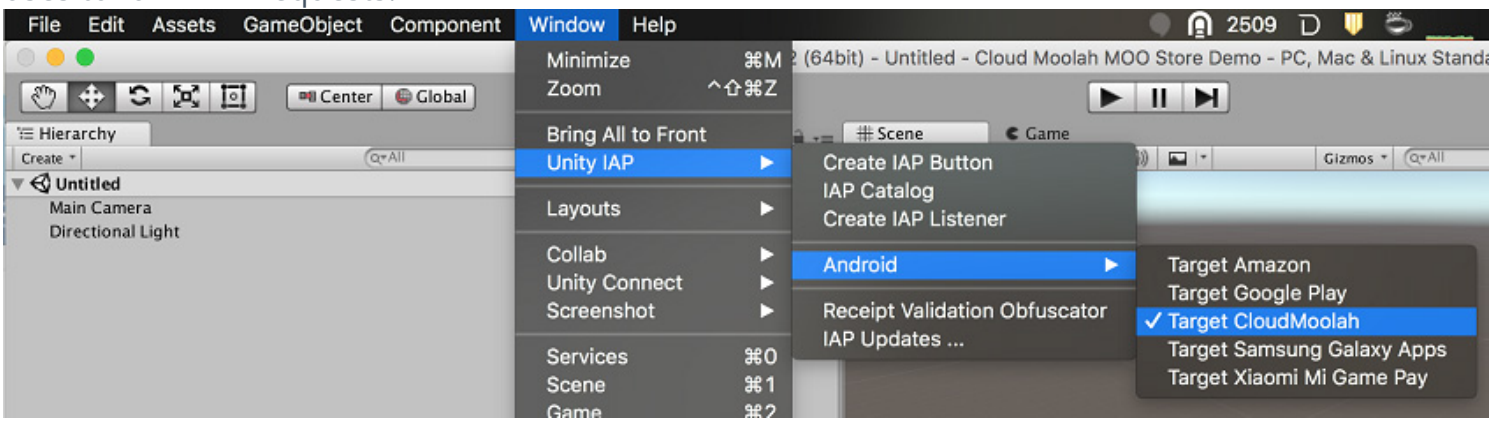
An app targeting the CloudMoolah MOO Store must configure the CloudMoolah appKey and hashKey using the IMoolahConfiguration via the scripting API before initializing **Unity IAP**. Collect these variables' values from the app's CloudMoolah developer dashboard.

The following example configures and starts initialization of Unity IAP:

```
using UnityEngine.Purchasing;
public class MyStore : IStoreListener
{
    public void InitializeStore()
    {
```

```
        var module = StandardPurchasingModule.Instance();
        var builder = ConfigurationBuilder.Instance(module);

        // Configure CloudMoolah
        builder.Configure<IMoolahConfiguration>().appKey = "d93f4564c41d463ed3d3
        builder.Configure<IMoolahConfiguration>().hashKey = "cc";
        // For server-to-server (also called "online" games) transaction
        // logging, set IMoolahConfiguration.notificationURL.
        builder.Configure<IMoolahConfiguration>().notificationURL = "https://gam
        builder.Configure<IMoolahConfiguration>().SetMode(CloudMoolahMode.Produc
        // Add purchasable products. The product must be defined in the store.
        // Unity IAP provides the *ProductType* enumeration to specify the durab
        // of a purchasable product. CloudMoolah limits the product type to Cons
        builder.AddProduct("100.gold.coins", ProductType.Consumable);

        // Start asynchronous IAP initialization.
        UnityPurchasing.Initialize(this, builder);
    }
}
```

In the Unity Editor, set the Unity IAP Android target to CloudMoolah by selecting **Window > Unity IAP > Android > Target CloudMoolah**. This sets CloudMoolah as the Android store that the game uses to fulfill IAP requests.



Enable CloudMoolah in the Editor

To set the target store in your initialization code, call the UnityPurchasingEditor.TargetAndroidStore function:

UnityPurchasingEditor.TargetAndroidStore(AndroidStore.CloudMoolah);

Build a signed non-Development Build Android **APK** from your game. For more information, see Getting started with Android development.

Note: Take special precautions to safely store your **keystore** file. The original **keystore** is always required to update a published app.

For more information on setting up and configuring IAP, see Setting Up Unity IAP, Unity In-App Purchasing Initialization, and Integrating Unity In-App Purchasing with your game.

Test IAP

The CloudMoolah MOO Store supports testing. To test your game in the CloudMoolah MOO Store, you must enable developer mode in the app before making purchases by calling the `IMoolahConfiguration.SetMode` function.

When you set the test mode in a build of your game, transactions are processed against a dummy offline store. This allows you to test the app's purchasing logic without incurring real-world monetary costs related to the product.

To modify the game's MOO Store test mode, create the ConfigurationBuilder instance and add the following line, then build and run the app to test its IAP logic:

```
// TESTING: auto-approves all transactions
builder.Configure<IMoolahConfiguration>().SetMode(CloudMoolahMode.AlwaysSucceed)
```

To test error handling, configure the test mode to fail all transactions. To do this, use the CloudMoolahMode.AlwaysFailed enumeration:

```
builder.Configure<IMoolahConfiguration>().SetMode(CloudMoolahMode.AlwaysFailed);
```

Note: When you finish testing, remove the SetMode statement that configures developer mode, or change the parameter to use the CloudMoolahMode.Production enumeration value. This ensures users pay real-world money when the app is in use.

Example implementation

Unity IAP's default integration includes an implementation script example that demonstrates how to use the required, and some of the optional, CloudMoolah scripting APIs.

To view the CloudMoolah-specific example:

On your development machine, open the folder for the app in which you have enabled IAP.

Open the ***Assets/Plugins****/UnityPurchasing/script* folder.

In the script folder, open *IAPDemo.cs*.

The example implements the `Awake` function, which calls the `IMoolahConfiguration` API to set `appKey` and `hashKey`. This connects your app to the CloudMoolah store server.

The *IAPDemo.cs* file also shows how to use optional API calls to configure test mode, and for extending functionality to include transaction restoration. For more information on the optional APIs, sign in to the

CloudMoolah Developer Portal website at dev.cloudmoolah.com.

2018–03–07 Page amended with limited editorial review

Unity IAP Xiaomi integration guide

Leave feedback

Overview

Unity IAP provides a streamlined channel for developers outside of China to publish apps in the Chinese market. This guide introduces the Unity Channel SDK, and describes the end-to-end process for a developer publishing IAP content to the Xiaomi Mi Game Center platform.

Publishing a game to the Mi Game Center is a 3-step process:

Submit your Xiaomi-configured game package through Unity Cloud Build.
In the Xiaomi-Unity developer portal, populate your game's store metadata.
Release your game upon Xiaomi approval.

Xiaomi Mi Game Center

The Mi Game Center is Xiaomi's official Android store. It allows users to search, browse and purchase products for the Xiaomi platform using a secure payment portal. For more information, visit Unity's Xiaomi partner site.

Unity Channel

Unity Channel is an internal **component** of Unity IAP that helps developers outside of China access the Chinese app store market by facilitating user login, payment management, and Chinese Government app distribution regulatory approval.

Because of Unity Channel, Xiaomi Mi Pay integration differs from Google Play and iTunes in the following ways:

Xiaomi games require a Unity IAP Catalog. Google Play and iTunes support entirely backend-defined Products. However, the Xiaomi client SDK built into Unity IAP depends on the Product IDs and metadata being available on the client.
Codeless IAP support is planned but not yet available (check forums for updates).
During Unity IAP initialization, Product ownership information is not returned to the client. Therefore, developers must track user purchases through their servers or locally on the device.

Requirements

The Unity Channel SDK is designed for use with Unity versions 2017.1+, however it is backwards compatible with versions 5.3+.
The Unity Channel SDK is included in Unity IAP versions 1.13.0+.
Xiaomi IAP support is only available for Android builds. The appropriate Android and Java SDKs are required.
Pushing builds to Xiaomi requires the Unity Cloud Build service.
Xiaomi does not currently support Codeless IAP implementation.

Building your Project for Xiaomi

This section describes the process of configuring your game through the Unity Editor using the Unity IAP SDK:

Setting up your Project
Adding the Xiaomi package to your Project
Enabling IAP
Creating an IAP Catalog

Setting up your Project

Configuring for Android

Xiaomi only supports Android builds. Configuring your Project for Android is a 4-step process.

Download and install the Android and Java SDKs. For more information, see documentation on Getting started with Android Development.
Set the SDK file path targets. In the Unity Editor, select **Edit** > **Preferences**, then select **External Tools** from the left navigation bar. Scroll down to the **Android** section, and set the **SDK** and **JDK** fields to reflect the file paths where

you installed the Android and Java SDKs respectively (see image 1.1, below; note that you can select the **Download** buttons next to each field for direct links to each SDK's download webpage).
Set the application's Package Name. Xiaomi requires the application's **Package Name** to follow the "BuildName.mi" convention. Select **Edit** > **Project Settings** > **Player** to open the Player Settings window. Then select the Android tab (see image below) and, under **Other Settings** > **Identification**, and set the **Package Name** field (see image 1.2, below).
Build the Project for Android. In the Editor, select **File** > **Build Settings**, then select **Android** from the menu. Select **Build** to test that Unity can locate the SDKs and compile your build.

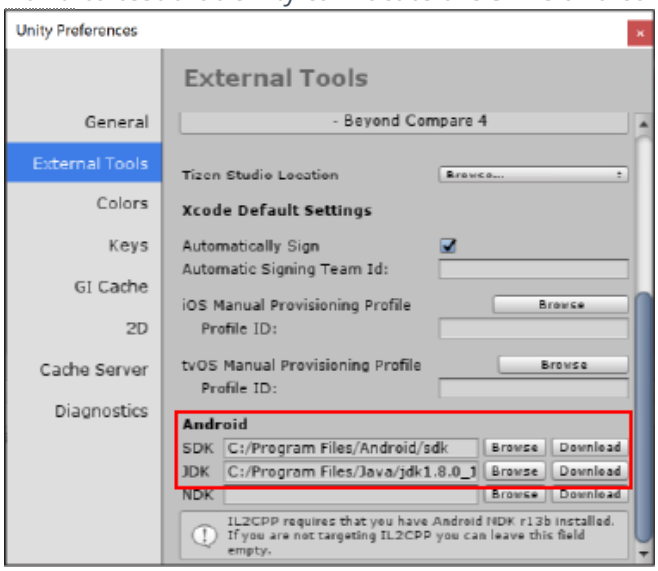


Image 1.1: Specifying file locations to the Android and Java SDKs in the Unity Editor.

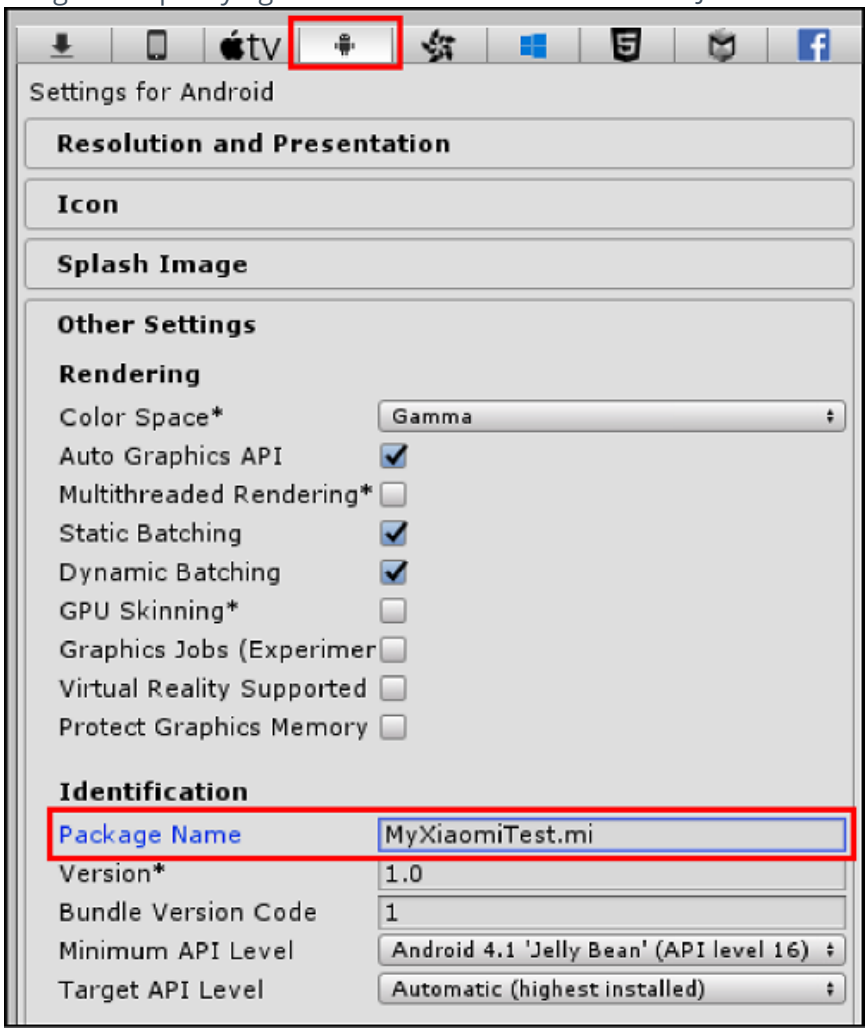
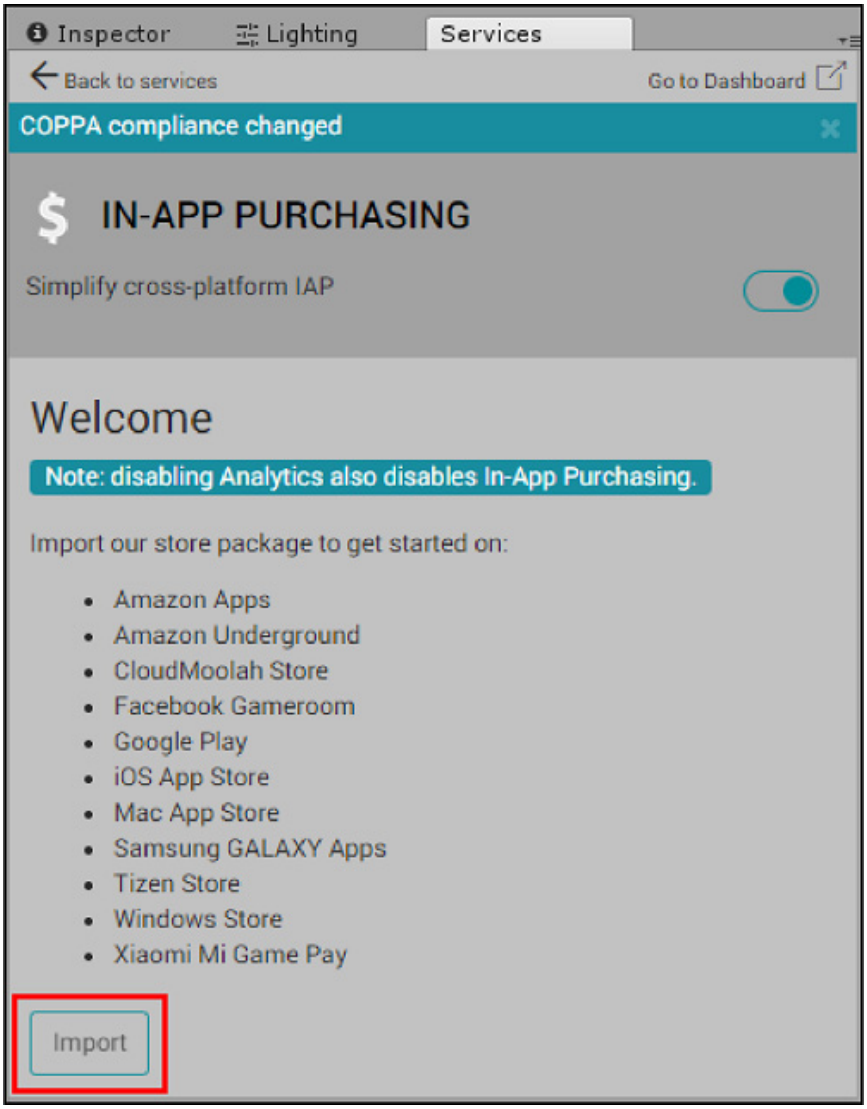


Image 1.2: Setting your app's package name in the Unity Editor Android settings.

Adding the Xiaomi Asset package to your Project

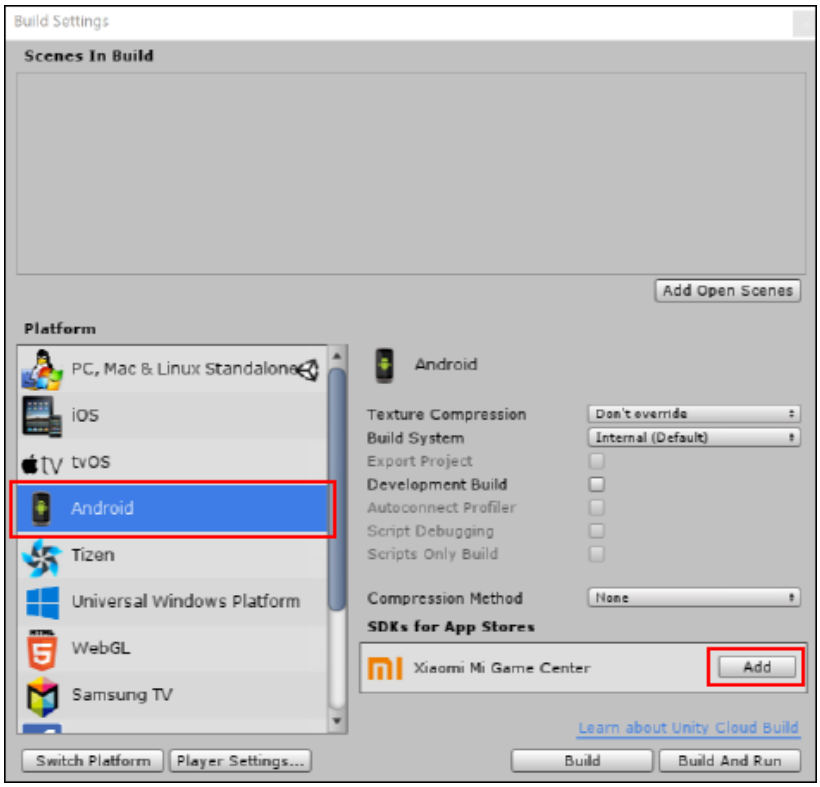
In the Editor, enable Unity IAP from the Services window (**Window** > **General** > **Services**; see documentation on Setting up Unity IAP). Be sure to Import the IAP Asset package when prompted (see image below). As of version 1.13.0, the Unity IAP Asset package includes the Unity Channel and Xiaomi SDKs.



Enabling Unity IAP and importing the Xiaomi SDK via the Unity Editor's Services window.

[Optional] Standalone SDK

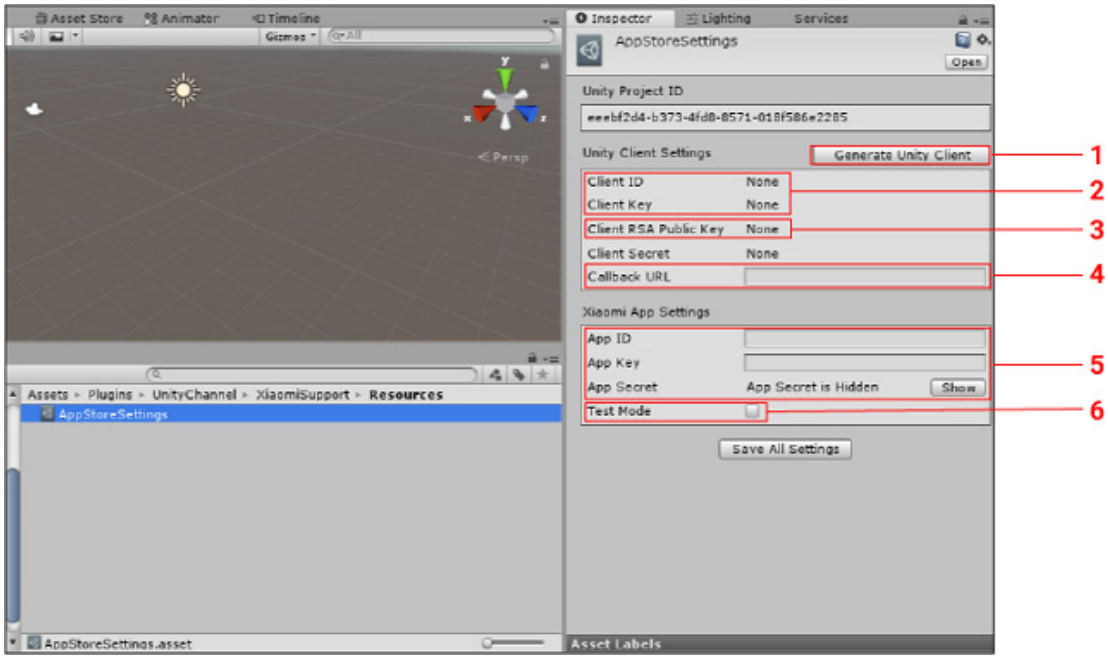
If you wish to publish your app to the Mi Game Center without in-app purchasing, install the Xiaomi Unity Channel standalone SDK from the Build Settings window (**File** > **Build Settings**) by selecting **Android** from the **Platform** menu and selecting **Add** from the **Xiaomi Game Center** menu option that appears.



Installing the Unity Channel standalone SDK from the Build Settings window.

App store settings

Unity Channel includes an Asset that provides an Editor interface for managing app store credentials and test mode functionality. This **AppStoreSettings** Asset installs to *Assets/Plugins/UnityChannel/XiaomiSupport/Resources* when you import the Unity IAP Asset package. You can also create it manually in the Editor by selecting **Assets** > **Create** > **App Store Settings**. Access the interface by selecting the Asset and viewing the Inspector.



The AppStoreSettings asset provides a GUI for managing your app's credentials.
Note: The **AppStoreSettings** Asset is only available in Unity 5.6+.

In order to communicate, the Unity and Xiaomi servers both require unique identifiers for your game. Note that you can retrieve your Unity Client credentials here by pasting your Project ID into the search field. Unity 5.3 or 5.4 users must retrieve and set

credentials this way, as they cannot access the **AppStoreSettings** Asset. The settings depicted in the image above are described below:

The **Generate Unity Client** button populates Unity credentials that bind to your Project.

Note: Generating, loading, or updating Unity client credentials sends networked requests to Unity's backend server. Progress for this operation appears in the Console log.

Client ID and **Client Key** are required to communicate through Unity IAP to the Unity Channel and Xiaomi backends. The Unity Channel backend proxies receipt validation requests (for more information, see documentation on Unity IAP receipt validation, and the **Receipt validation and extensions** section of the Unity Channel SDK documentation).
Client RSA Public Key is an optional security layer. Use it to receive server callbacks for client-side receipt validation, or to integrate with Unity server APIs.

Note: **Client Secret** is another optional security layer should you wish to implement server-side validation. **Client Secret** is auto-generated with the other credentials. Refresh it any time by selecting **Update Client Secret** from the **AssetStoreSettings** Asset.

Provide an optional **Callback URL** for your game server to receive purchase transaction data.
Xiaomi distributes **App ID**, **App Key**, and **App Secret** credentials during the submission process. You must save them to your App Store Settings before submitting your final build for publication.
Test Mode is required for testing your application's purchase flows and submitting to Xiaomi.
You need these credentials and the test mode toggle at various points of the integration process. Note that App Store Settings data updates server-side, as opposed to saving to your client.

Implementing IAP

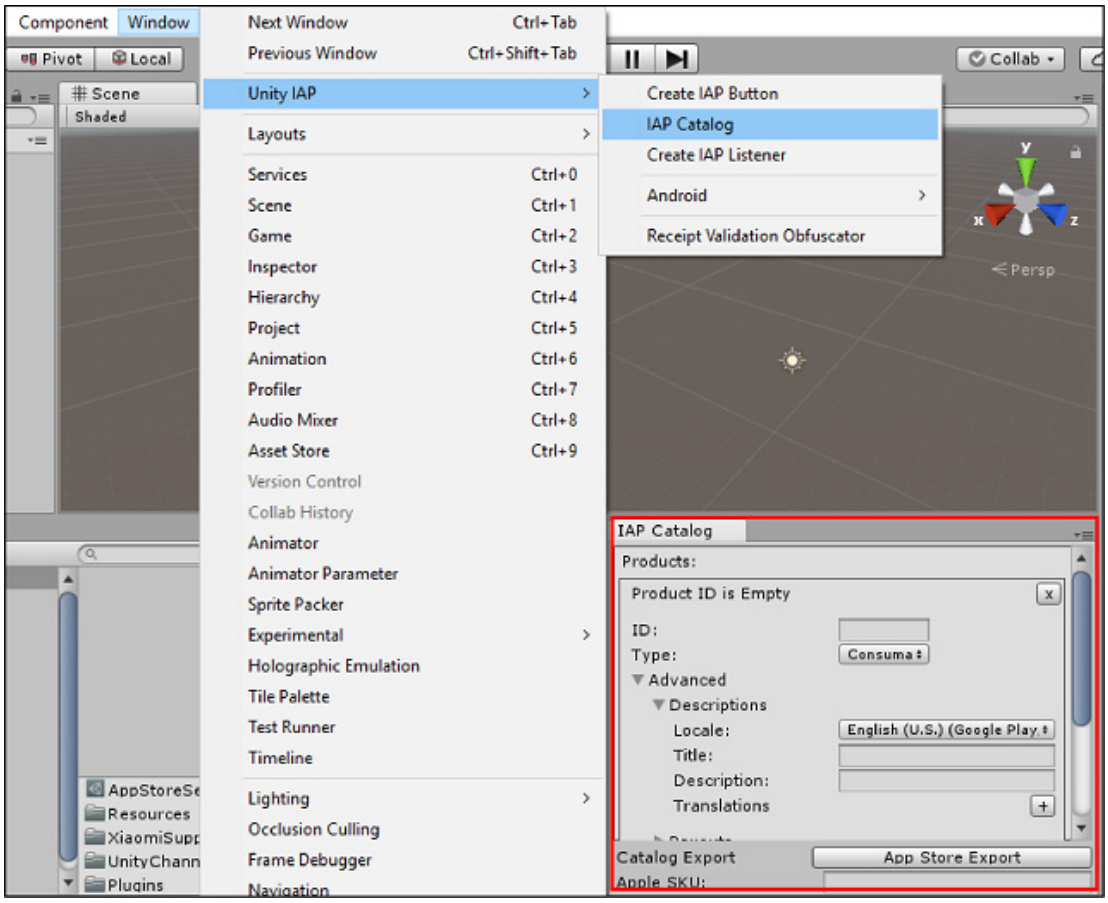
Enabling the Unity IAP service

If you enabled Unity IAP to import the Xiaomi Asset package, no action is required. Otherwise, see documentation on Setting up Unity IAP to enable the service.

You can also install the latest Unity IAP **plugin** through the Unity Asset Store.

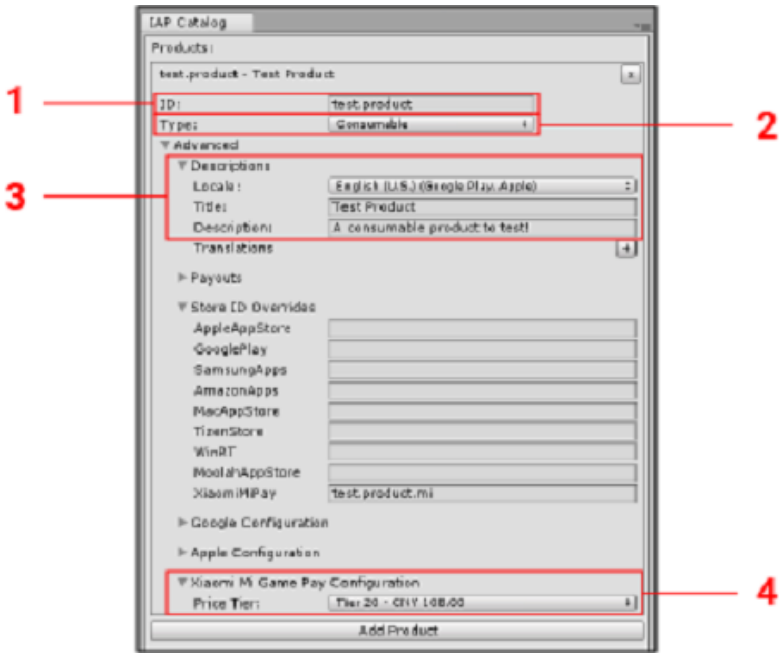
Creating a Product Catalog

The Xiaomi client SDK built into Unity IAP depends on Product metadata being available in the client. As such, you must define your Products through the Editor's IAP Catalog GUI (**Window** > **Unity IAP** > **IAP Catalog**).



The IAP Catalog window provides a GUI for building and exporting your app's Product catalog.

Populating the Catalog The IAP Catalog GUI defines Products along with their metadata for the game client's runtime. Documentation on Codeless IAP contains information on configuring Products through the IAP Catalog. The following Product attributes warrant attention for Xiaomi specifically:



Breakdown of the IAP Catalog editor.

ID is a unique identifier (string data type) required by the Mi Game Center.

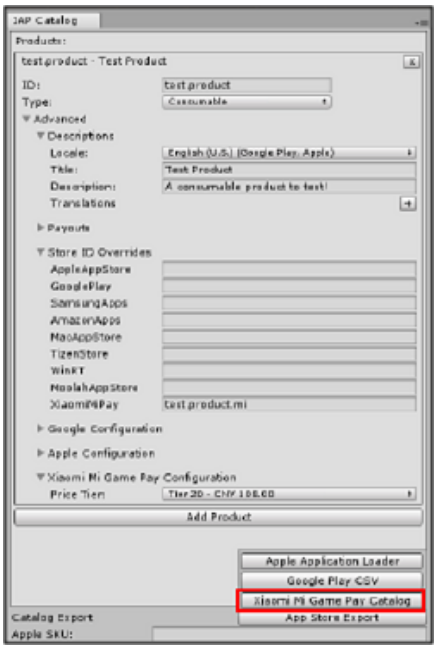
Type indicates the durability of the Product. Xiaomi supports consumable and non-consumable Product Types. It does not support subscription Product Types.

The **Descriptions** section controls **Title** and **Description** text for use on the Mi Store. You must specify **Simplified Chinese** from the **Locale** drop-down menu (this is currently the only supported language).

Pay Configuration controls the Product's price point. The Unity Channel SDK supports predefined Chinese yuan pricing tiers for Xiaomi, but not arbitrary values.
For more information about Product attributes and their parameters, see documentation on Defining Products.

Exporting the Catalog for Xiaomi's developer portal

In the Editor, in the IAP Catalog window, export the Catalog by selecting **App Store Export** > **Xiaomi Mi Pay Catalog**.

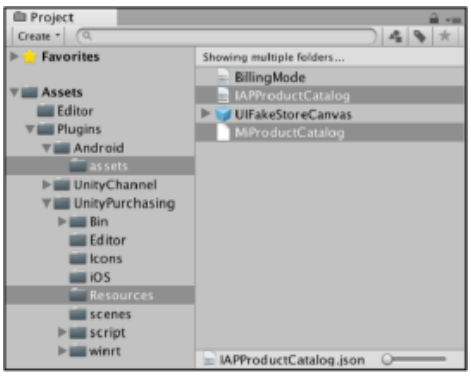


Exporting your app's Product catalog for the Xiaomi developer portal.

Export the *MiGameProductCatalog.prop* file to your location of choice. Import your Product catalog on the Xiaomi developer portal by navigating to your Project's **IAP Configuration** tab and selecting **Import** (see section on Importing your IAP Product catalog, below).

Example of the Xiaomi developer portal reading the *MiGameProductCatalog.prop* file for its interface.

Exporting the IAP Catalog also writes a copy of the *MiGameProductCatalog.prop* file to your Project's *Assets/Plugins/Android/assets/* directory (see image below). Unity IAP uses this file in the IAP Catalog editor and at runtime. It also serves as a default Product catalog for your app if no catalog file is explicitly imported through the Xiaomi developer portal.



The Product catalog exported.

Client-side initialization

Unity IAP typically requires a configuration builder that consumes IAP Catalog data to be parsed for the destination store, followed by the Unity IAP `Initialize()` API (see documentation on Initializing IAP).

Xiaomi games require an extra step prior to initializing IAP, because the Mi Game Center requires apps to share their credentials through the login API at launch.

The modified initialization process becomes:

Initialize the Unity Channel Xiaomi API.
Log in to the Unity Channel Xiaomi API.
Initialize Unity IAP as normal by running a configuration builder instance, then call the Unity IAP `initialize()` API.

Execute these steps early in the game's runtime lifecycle, preferably at launch. You can implement them in the same script.

1. Xiaomi Login

The following example illustrates how to modify initialization using the Unity Channel SDK to call the login API:

```
using AppStoreSupport;
using System;
using System.Collections;
using UnityEngine;
using UnityEngine.Purchasing;
using UnityEngine.Store;

// Run the script once, at game launch.
public class UnityChannelSample : MonoBehaviour
{
    // Xiaomi login requires a login listener.
    // Create one by implementing the ILoginListener abstract class included in the Unity Char
    class SampleLoginListener : ILoginListener
    {
        public Action initializeIAP;

        // Step 1 succeeded; call step 2
        public void OnInitialized()
        {
            Debug.Log("Initialization succeeded.");
            UnityEngine.Store.StoreService.Login(this); // If initialization succeeds, initiat
        }

        // Step 1 failed; display error message and recourses
        public void OnInitializeFailed(string message)
        {
            Debug.Log("Initialization failed.");
        }

        // Step 2 completed; call step 3
        public void OnLogin(UserInfo userInfo)
        {
            Debug.Log(string.Format("Login successful: userId {0}, userLoginToken {1}, channel
            // When login succeeds, proceed to initializing IAP
initializeIAP();
        }

        // Step 2 failed; display error message and recourses
        public void OnLoginFailed(string message)
        {
            Debug.Log("Login failed.");
```

```
        }
    }
}
```

Note: You must use all of the functions in the above sample code, which account for all possible steps of the login process. However, you can choose the actions called within each function. In this example, debug text appears in place of action calls. In this example, the OnLogin() method initiates a Unity IAP initialization function.

2. Xiaomi initialization

Next, initialize the Xiaomi API using a login listener and the app credentials stored in your **AppStoreSettings** Asset:

```
    void Awake()
    {
        // Create a login listener based on the SamleLoginListener class
        SampleLoginListener loginListener = new SampleLoginListener();

        // Tie the initializeIAP Action (from the SampleLoginListener class) to a function (de
        loginListener.initializeIAP = ConfigureIAP;

        // Access the AppStoreSettings Asset to generate AppInfo with your stored credentials.
        AppStoreSettings appStoreSettings = Resources.Load<AppStoreSettings>("AppStoreSettings

        // Initialize Xiaomi using the credentials and login listener
        UnityEngine.Store.StoreService.Initialize(appStoreSettings.getAppInfo(), loginListener
    }
```

Upon loading your Project's **AppStoreSettings** Asset, the `getAppInfo()` function returns credential data for the Mi Game Center store initialization process.

3. Unity IAP configuration builder and initialization

Finally, use a configuration builder to translate Product data from your IAP Catalog for Xiaomi, then initialize Unity IAP:

```
    // The configuration builder requires a store listener.
    // Create one by implementing the IStoreListener abstract class.
    class StoreListener : IStoreListener
    {
        private IStoreController controller;
        private IExtensionProvider extensions;

        // Called when Unity IAP is ready to make purchases.
        public void OnInitialized(IStoreController controller, IExtensionProvider extensions)
        {
            this.controller = controller;
            this.extensions = extensions;
        }

        // Note that this will not be called if Internet is unavailable;
        // Unity IAP will attempt initialization until it becomes available.
```

```
    public void OnInitializeFailed(InitializationFailureReason error)
    {
    }

    // Called when a purchase completes, or Unity IAP encounters an unrecoverable initiali
    // may be called at any time after OnInitialized().
    public PurchaseProcessingResult ProcessPurchase(PurchaseEventArgs e)
    {
        return PurchaseProcessingResult.Complete;
    }

    // Called when a purchase fails.
    public void OnPurchaseFailed(Product i, PurchaseFailureReason p)
    {
    }
}

// Run a configuration builder to define your Products
public void ConfigureIAP()
{
    // Load Products from the IAP Catalog
    var module = StandardPurchasingModule.Instance();
    var builder = ConfigurationBuilder.Instance(module);
    var catalog = ProductCatalog.LoadDefaultCatalog();

    // Loop through Products in Catalog, extracting metadata to the builder
    foreach (var product in catalog.allProducts)
    {
        if (product.allStoreIDs.Count > 0)
        {
            var ids = new IDs();
            foreach (var storeID in product.allStoreIDs)
            {
                ids.Add(storeID.id, storeID.store);
            }
            builder.AddProduct(product.id, product.type, ids);
        }
        else
        {
            builder.AddProduct(product.id, product.type);
        }
    }

    // Create a store listener based on the StoreListener class
    StoreListener storeListener = new StoreListener();

    // Initialize Unity IAP
    UnityPurchasing.Initialize(storeListener, builder);
}
```

The configuration builder references Product IDs and Product Types to automate population of Price, Type, Title, and Description metadata. It uses your IAP Catalog's *IAPProductCatalog.json* file and the `builder` object instantiated during Unity IAP initialization.

Implementing IAP purchasing

For information on methods for in-game purchase implementation, see documentation on the following:

Codeless IAP
Unity IAP Initialization
Unity Channel SDK

Testing integration

Test your game's purchase flows on a non-Xiaomi Android device by toggling test mode in the **AppStoreSettings** Asset interface. Unity Channel also provides an API for toggling test mode:

```
AppInfo.debug = true;
```

The Unity Channel wrapper includes the `AppInfo` class, and passes data to the Xiaomi SDK. To submit your app to Xiaomi, you must enable test mode. Test mode also bypasses the credit card requirement for testing purposes. After enabling test mode, build your Project and launch the resulting APK file from an Android device.

To test real purchasing, set test mode to false and test on a Xiaomi device. Please note that real purchasing requires a Xiaomi Mi Pay Account and appropriate currency such as Chinese Yuan.

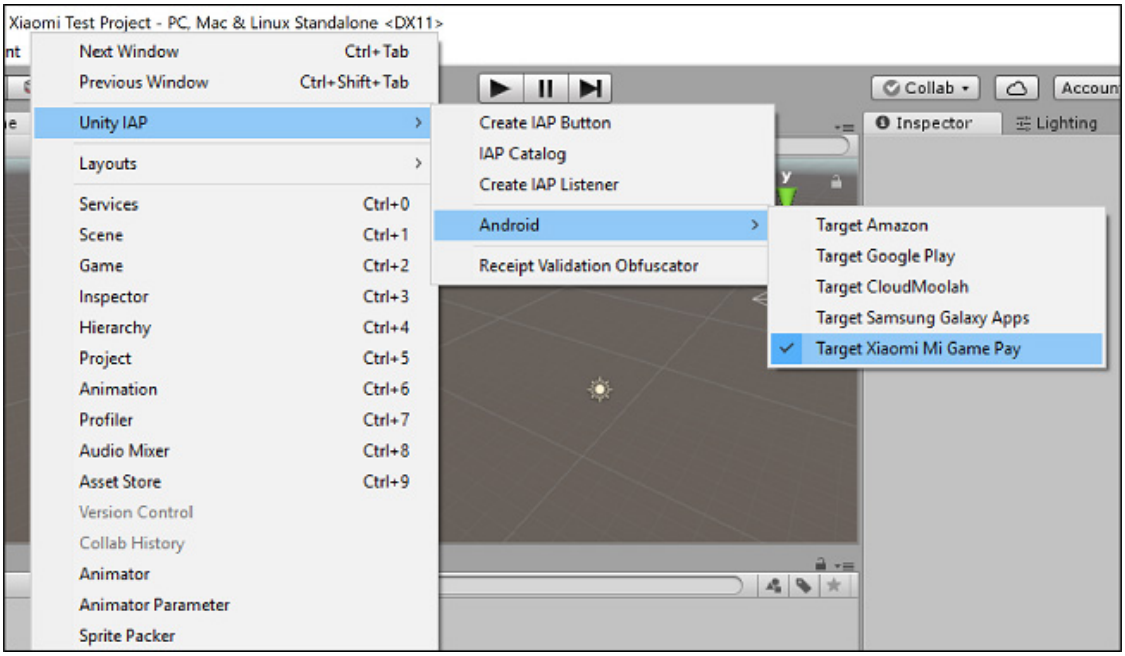
Submitting to Xiaomi

This section describes the process of submitting your game through the Unity Editor using the Unity IAP SDK:

Set the IAP target
Test IAP integration
Register the application
Push your game to the Xiaomi portal

Setting the IAP target

In the Unity Editor, set Unity IAP to target Mi Game Pay by selecting **Window** > **Unity IAP** > **Android** > **Target Xiaomi Mi Game Pay**. This enables Xiaomi for the next build of the game's APK. It also creates the configuration file that informs Unity IAP at runtime to use the Xiaomi Mi Pay native billing API.



Setting Unity IAP to target Xiaomi Mi Game Pay during the next construction of your app's build.

Pushing the Project to the Xiaomi developer portal

Before pushing to Xiaomi, test building your game, either locally or by configuring the Project with an Android build target in Unity Cloud Build (see documentation on Cloud Build).

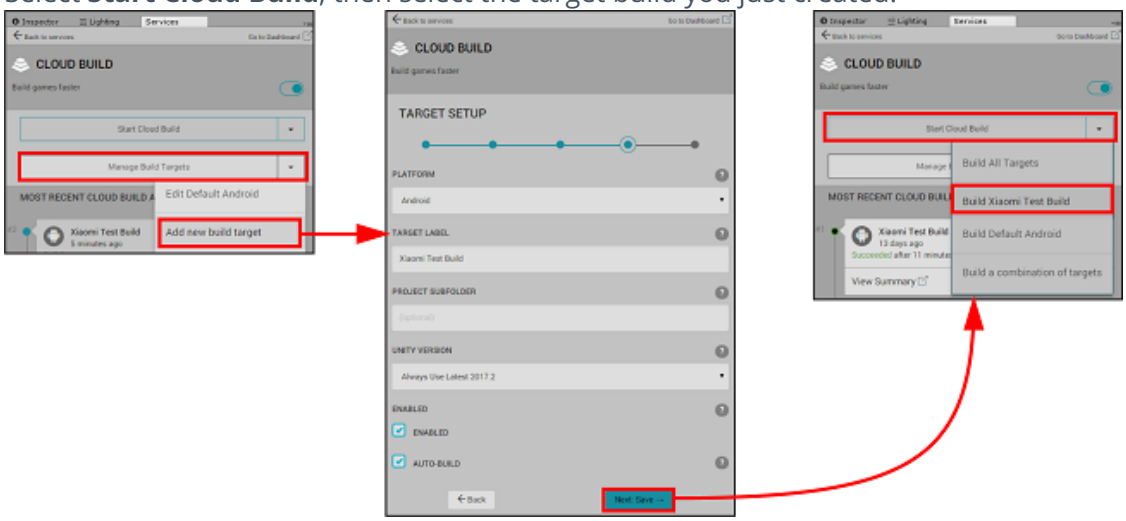
In the Editor, enable Cloud Build through the Unity Services window (see documentation on Cloud Build implementation).

Upload the build

Upload your build to the build history for your project, using one of two methods:

From the Editor:

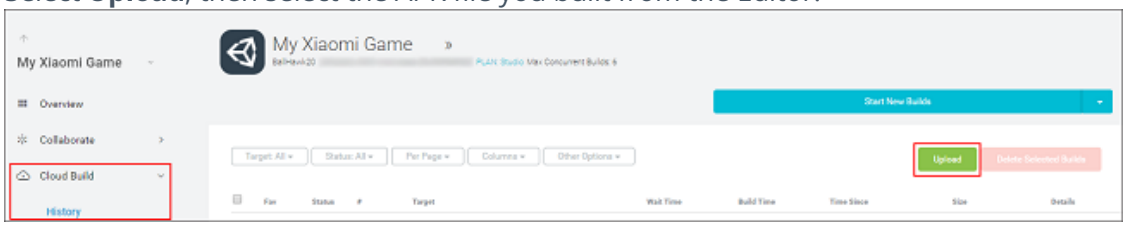
In the Cloud Build Services window, select **Manage Target Builds** > **Add New Target**.
In the **TARGET SETUP** menu, set the **PLATFORM** field to **Android** and enter a useful **TARGET LABEL**. Then select **Next: Save**.
Select **Start Cloud Build**, then select the target build you just created.



Uploading a Cloud Build via the Editor.

From the Unity Cloud Build Developer Dashboard:

Navigate to your Project's **Cloud Build History**.
Select **Upload**, then select the APK file you built from the Editor.



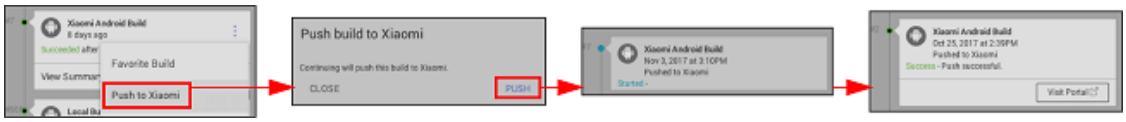
Uploading a Cloud Build via the Developer Dashboard.

Push the build to Xiaomi

Push the the hosted build to Xiaomi's developer portal, using one of two methods:

From the Editor:

In the Cloud Build Services window, locate the desired build from the build history timeline and select **Push to Xiaomi**. Verify that you want to push, and that the action completes.



Pushing a hosted build to Xiaomi's developer portal via the Editor.

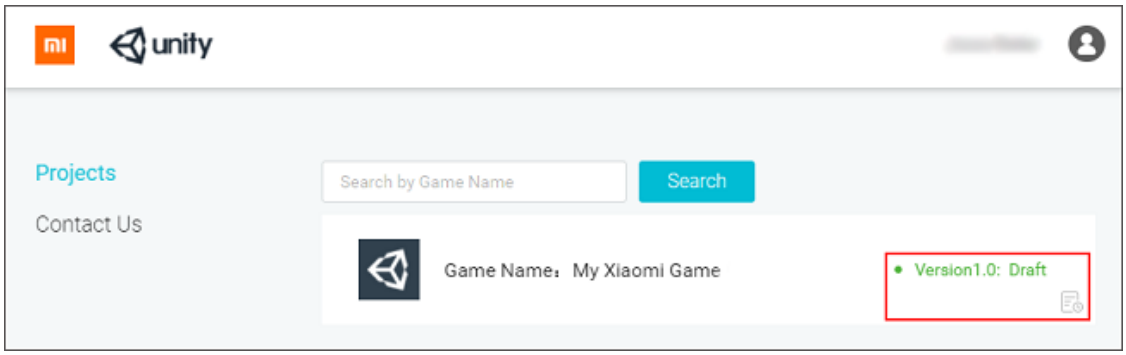
From the Unity Cloud Build Developer Dashboard:

Navigate to your Project's **Cloud Build History**.
Select **Download .APK** file, then select **Push to Xiaomi** from the dropdown menu.

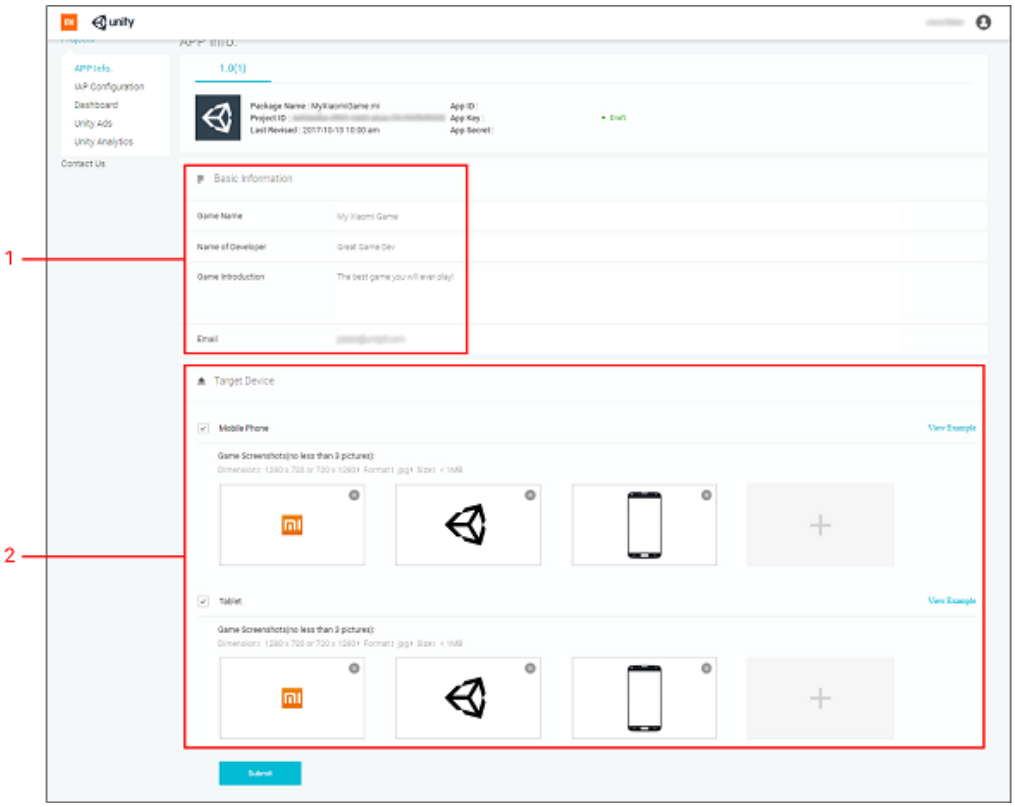
Pushing a hosted build to Xiaomi's developer portal via the Unity Developer Dashboard.
Note: Only the owner of the organization that your project is under will be able to deploy to Xiaomi.

Configuring the game in the Xiaomi developer portal

Your UDN credentials grant access to your uploaded Projects in the Xiaomi Unity developer portal. The first time you log in, you must acknowledge Xiaomi's Terms and Conditions before proceeding to your **Projects** list. Locate the Project you wish to submit. Its status (highlighted below) will initially read **Version1.0: Draft**. Select the clipboard icon to view a submission log of your Project's status changes throughout the process. Select the status link to expand your Project's metadata details.



Your Projects list in the Xiaomi developer portal.
The following details are illustrated in the image below:



A Project's store metadata expanded in the Xiaomi developer portal.
1. Basic Information

This section contains the text to display on the Mi Game Center.

Game Name is the English title of your app. Xiaomi will manually translate the English title to Chinese for the customer-facing storefront.
Name of Developer is your customer-facing publisher name.
Game Introduction is the English description of your app. Note that Xiaomi will manually translate the English title to Chinese for the customer-facing storefront.
Email is the developer contact information Xiaomi should use to initiate feedback.

2. Target Device

This section contains marketing assets for Phone and Tablet devices that display on the Mi Game Center. Note the following guidelines:

You must upload a minimum of 3 images for each device type that you check.
Image dimensions must either be 1080 x 720 pixels (landscape), or 720 x 1080 pixels (portrait).

Submitting for review

When you are ready, select **Submit** to submit your game to Xiaomi for review. The status indicator for your Project changes to **In review**. While your Project is in review, you cannot edit its details. Expand the Project and select **Cancel** to cancel the review and revise the Project's details.

Regional compliance with regulations

You need an ISBN publication license to publish your app. Xiaomi provides comprehensive support for the ISBN approval process through the State Administration of Press, Publication, Radio, Film and Television (SAPPRFT).

Xiaomi does not guarantee that the SAPPRFT will approve all games (see SAPPRFT submission guidelines). To meet store and country compliance rules, you may need to make some changes to your game.

Content guidelines for a smooth approval process

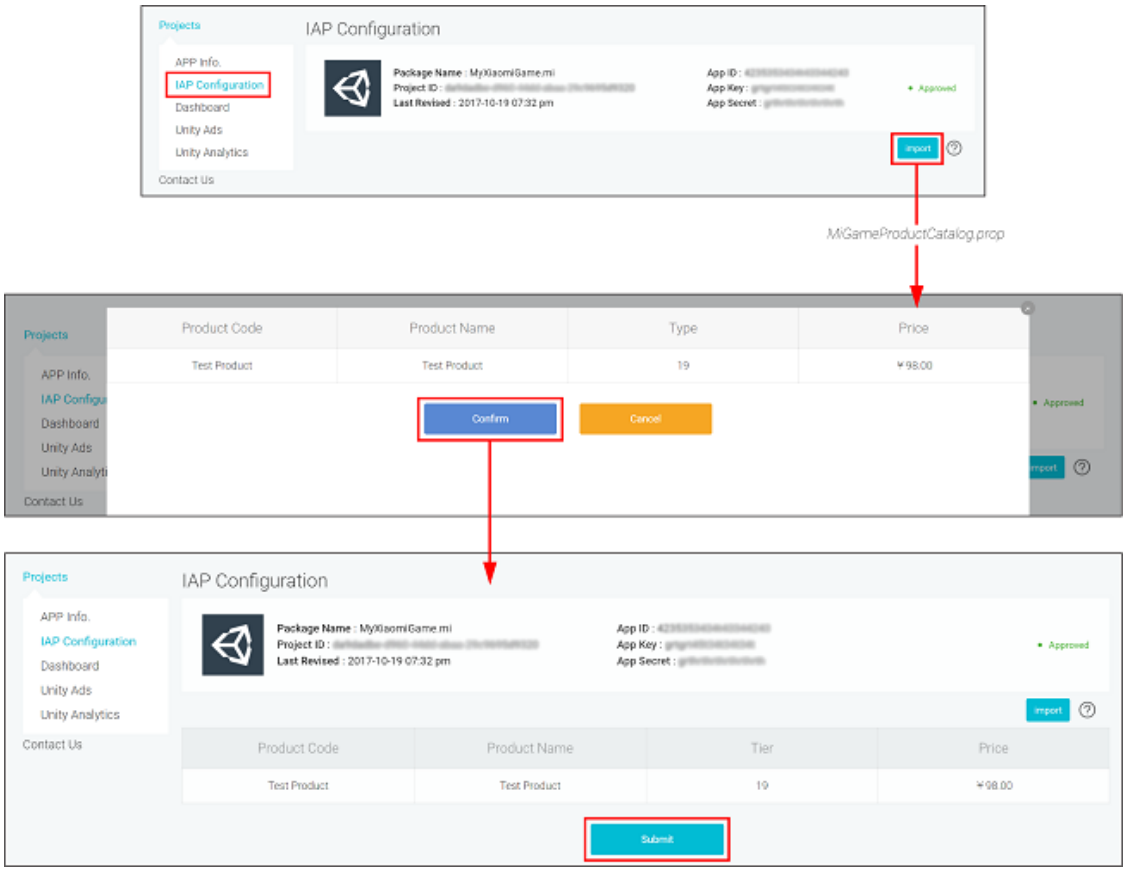
Consider the following tips when submitting for Chinese government approval:

Avoid depicting Chinese national landmarks and symbolism (for example, the Chinese national flag, or the Great Wall) in a manner that could be considered undignified.
Avoid depicting Chinese national politics, history, conflicts, or religions entirely.
When creating maps and regions in your game's world, avoid defining regions that conflict with China's existing geopolitics and borders.
Avoid depiction of gambling, and gambling mechanics of any kind.
China has tight restrictions on mature content. Avoid depictions of nudity, realistic violence, and drug usage.
Avoid depicting minors participating in any illegal activities, such as underage drinking and smoking.

Importing your IAP Product catalog

Before finalizing your submission, follow these steps to ensure Xiaomi has your most current IAP Product catalog:

Select your Project, then select **IAP Configuration** from the left navigation bar.
Select **Import**.
Import the *MiGameProductCatalog.prop* file you exported from your Project.
Verify the contents of the catalog, then select **Confirm**.
A list of your imported Products appears. Select **Submit** to save the catalog to your submission.



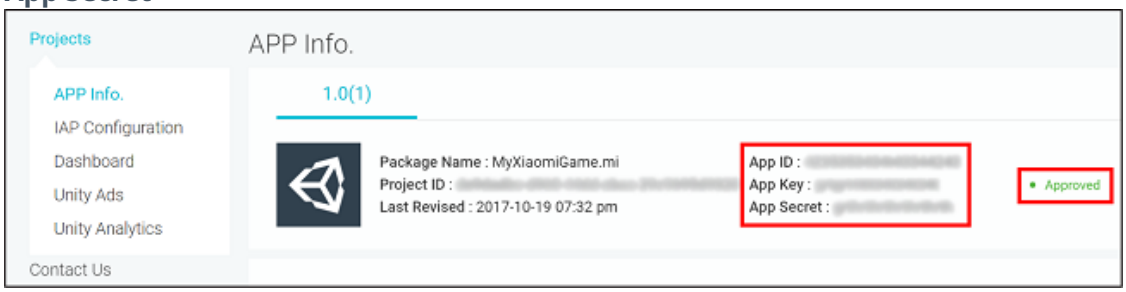
Importing an IAP catalog to the Xiaomi developer portal.

Publishing to the Mi Game Center

Credentials

When Xiaomi indicates approval, your project status changes to **Approved** and receives the following credentials:

App ID
App Key
App Secret



An approved Project on the Xiaomi developer portal.

Enter these credential values in the **AppStoreSettings** Asset or initialization script, then set test mode to `false`. Create a new build of your game, and push this build to Xiaomi as before.

Xiaomi provides a developer contract via email. Once you sign the contract, Xiaomi facilitates the government approval process. It will notify you when the game receives an ISBN license and clearance to publish on the Mi Game Center.

Xiaomi's content review process typically takes 1–2 business days, unless it recommends changes. The government approval process and license distribution, however, can take 4–8 months. For more information on the review process, please see the **FAQs** section of the Unity-Xiaomi partner page.

Unity Channel SDK and API extensions

Leave feedback

Unity Channel API

UnityEngine.Store namespace

AppInfo class

The `AppInfo` class stores client and Xiaomi credentials needed for client-to-server communication and authentication.

Key:	Type:	Description:
appID	string	Xiaomi app ID
appKey	string	Xiaomi app key
clientID	string	Unity client ID
clientKey	string	Unity client key
debug	bool	Toggle debug mode

ILoginListener interface

Xiaomi requires login through the `ILoginListener` for all apps publishing to the Mi Game Center.

Key:	Type:	Description:
OnInitialized ()	void	Called when initialization succeeds
OnInitializedFailed (string message)	void	Called when initialization fails
OnLogin ()	void	Called when login succeeds
OnLoginFailed (string message)	void	Called when login fails

StoreService class

Unity IAP uses the `StoreService` class internally to initialize the Unity Channel SDK.

Key:	Type:	Description:
static Initialize (AppInfo appInfo, ILoginListener loginListener)	void	Initialize the Unity Channel SDK
static Login (AppInfo appInfo, ILoginListener loginListener)	void	Login to the Xiaomi account

UserInfo class

A successful login to Unity Channel returns the `UserInfo` class. The data returned is for informational purposes.

Key:	Type:	Description:
channel	string	Indicates which channel you are using (currently, it is only “XIAOMI”)
userID	string	Channel’s unique user ID
userLoginToken	string	Validate the user login (see Server-side initialization section of Xiaomi IAP integration guide)

UnityEngine.ChannelPurchase namespace

IPurchaseListener interface

Unity IAP uses the `IPurchaseListener` interface internally to handle purchasing activities.

Implement the `IPurchaseListener` class to handle states of the purchase flow:

```
using UnityEngine.ChannelPurchase;
......
private class ExamplePurchaseListener : IPurchaseListener{
    public void OnPurchase (PurchaseInfo purchaseInfo){
        Debug.Log ("Purchase Succeed: " + purchaseInfo.gameOrderId);
    }
    public void OnPurchaseFailed (string message, PurchaseInfo purchaseInfo){
        Debug.Log ("Purchase Failed: " + message);
    }
    public void OnPurchaseRepeated(string productCode){
        Debug.Log ("Purchase Repeated");
    }
    public void OnReceiptValidate (ReceiptInfo receiptInfo){
        Debug.Log ("Validate Succeed");
    }
    public void OnReceiptValidateFailed (string gameOrderId, string message){
        Debug.Log ("Validate Failed");
    }
    public void OnPurchaseConfirm (string gameOrderId){
        Debug.Log ("Confirm Succeed");
    }
    public void OnPurchaseConfirmFailed (string gameOrderId, string message){
        Debug.Log ("Confirm Failed");
    }
}
```

Key:	Type:	Description:
OnPurchase ()	void	Called when purchase succeeds
OnPurchaseFailed (string message, PurchaseInfo purchaseInfo)	void	Called when purchase fails
OnPurchaseRepeated (string productCode)	void	Called when non-consumable Products are re-purchased
OnReceiptValidation (ReceiptInfo receiptInfo)	void	Called when `PurchaseService.ValidateReceipt(...)` succeeds
OnReceiptValidationFailed (string gameOrderID, string message)	void	Called when `PurchaseService.ValidateReceipt(...)` fails
OnPurchaseConfirm (string gameOrderId)	void	Deprecated
OnPurchaseConfirm (string gameOrderId, string message)	void	Deprecated

PurchaseService class

Unity IAP uses the `PurchaseService` class internally to initiate purchasing activities.

With the purchase listener implemented, call the `PurchaseService.Purchase` method to execute a purchase transaction:

```
using UnityEngine.ChannelPurchase;
......
    var myPurchaseListener = new ExamplePurchaseListener ();
    PurchaseService.Purchase ("Product ID", "Game Order ID", myPurchaseListener)
```

Product IDs are listed in the Xiaomi developer portal. If your app runs in debug mode, Product IDs also export to the *MiGameProductCatalog.prop* file. Game Order ID can be null; the Unity Channel SDK will generate its UUID.

Key:	Type:	Description:
static Purchase (string productCode, string gameOrderId, IPurchaseListener productListener)	void	Purchase Products
static ValidateReceipt (string gameOrderId, IPurchaseListener purchaseListener)	void	Validate purchase
static ConfirmPurchase ()	void	Deprecated (use `ValidateReceipt (...)` instead)

PurchaseInfo class

After a purchase, validate it by calling the `PurchaseService.ValidateReceipt` method:

```
    PurchaseService.ValidateReceipt(gameOrderId, myPurchaseListener);
```

If `gameOrderId` is valid, `myPurchaseListener` receives `ReceiptInfo` containing `signData` and `signature`, which Unity Channel uses to validate the purchase.

Key:	Type:	Description:
productCode	string	A Product's unique ID (you can also configure this directly in the Xiaomi dev portal)
gameOrderId	string	The order's purchase ID
orderQueryToken	string	Used to validate purchases

ReceiptInfo class

Validation usually occurs on the game server, however you can also validate the `signData` client-side.

Key:	Type:	Description:

Key:	Type:	Description:
gameOrderId	string	The order's purchase ID
signData	string	A JSON string containing details of the purchase
signature	string	signData's signature

Receipt validation and extensions API

Before continuing, please review documentation on Unity IAP receipt validation. Also see the UnityEngine.Purchasing Scripting API.

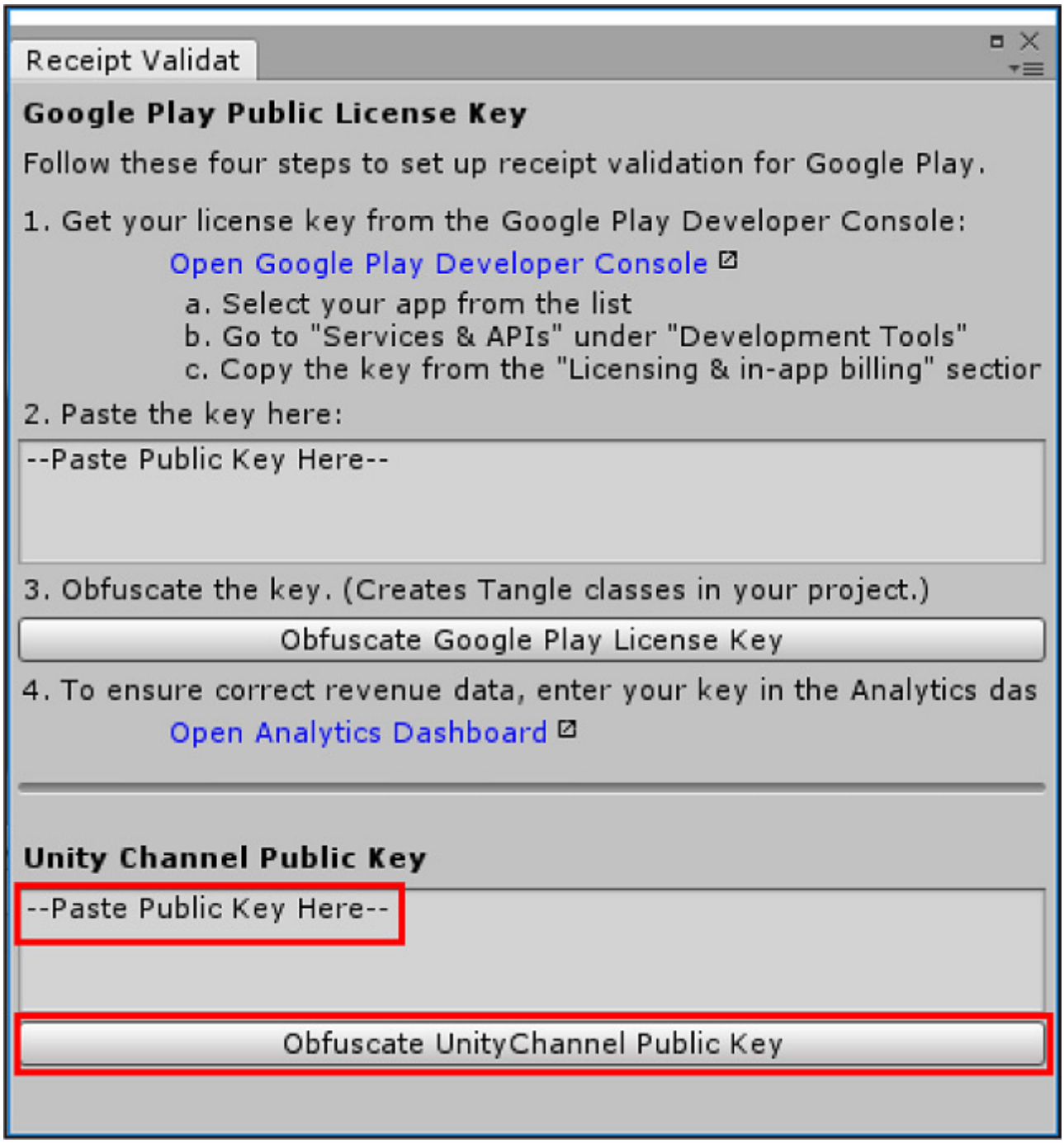
UnityEngine.Purchasing namespace

The `IUnityChannelConfiguration` and `IUnityChannelExtensions` interfaces offer extended functionality for unpacking app store receipts.

Perform local receipt validation by using the Unity IAP `CrossPlatformValidator` class (detailed in the Receipt validation documentation) and the `IUnityChannelExtensions.ValidateReceipt` method (detailed below) at the time of purchase.

Before implementing validation, enable receipt obfuscation for Xiaomi:

Obtain your app's **Client RSA Public Key** from your App Store Settings (see the **App Store Settings** section of the Xiaomi IAP integration guide), or from the Unity Client Settings dashboard.
In the Unity Editor, enter the Unity Channel key into the Unity IAP Receipt Validation Obfuscator window (**Window** > **Unity IAP** > **Receipt Validation Obfuscator**). This window collects, obfuscates, and stores public keys in the Project for receipt validation purposes.
In the Receipt Validation Obfuscator window, select **Obfuscate Unity Channel Public Key** to save the resulting *UnityChannelTangle* datafile to the Project.



The Unity IAP Receipt Validation Obfuscator window. Generates data classes in the project for use with the CrossPlatformValidator.

IUnityChannelConfiguration interface

To automatically fetch the required receipt data from Unity Channel during purchase, set `IUnityChannelConfiguration.fetchReceiptPayloadOnPurchase` to `true` when initializing Unity IAP. This calls `IUnityChannelExtensions.ValidateReceipt` internally after a successful purchase (ideally when the `PurchaseEventArgs`' `purchasedProduct.receipt` field triggers the `IStoreListener.ProcessPurchase` success callback).

Note: Setting the `fetchReceiptPayloadOnPurchase` flag to true incurs a network request to fetch the cryptographic receipt data after a successful purchase. If this second network request fails for any reason, the `ProcessPurchase` success callback receives a Product with an invalid receipt.

Passing `purchasedProduct.receipt` data through the `CrossPlatformValidator.Validate` API returns an `IPurchaseReceipt`, which is the fully validated receipt. If the result is empty, the validation failed. See the example below:

```
CrossPlatformValidator validator = new CrossPlatformValidator (GooglePlayTangle.

var result = validator.Validate(purchasedProduct.receipt);
```

Key:	Type:	Description:
fetchReceiptPayloadOnPurchase	bool	Automates Unity Channel receipt data collection during purchase

IUnityChannelExtensions interface

Use the `IUnityChannelExtensions` interface to confirm purchases and validate receipts. The callbacks for the above methods return a bool indicating success, a 'signData' string, and a 'signature' string (see section on ChannelPurchase.ReceiptInfo).

Note: `ValidateReceipt ()` is vulnerable to man-in-the-middle attacks; use the Client RSA Public Key for added security (see the **App Store Settings** section of the Xiaomi IAP integration guide).

Key:	Type:	Description:
`ConfirmPurchase (string transactionID, Action<bool, string, string>, callback)`	void	Collects purchase status for a prior purchase (used to review purchase history, especially for game interruption or network timeouts)
`ValidateReceipt (string transactionID, Action<bool, string, string>, callback)`	void	Validates the receipt for a given transactionID

UnifiedReceipt class

The `UnifiedReceipt` class contains store-specific transaction data that Unity IAP can interpret. See the example below on how to unpack a unified receipt string:

```
var unifiedReceipt = JsonUtility.FromJson<UnifiedReceipt>(purchEvtArg.purchasedP
```

Key:	Type:	Description:
Payload	string	Unity IAP's wrapper for receipts that come back in different formats for different stores
Store	string	The store on which the purchase occurred
TransactionID	string	The unique transaction ID

UnityChannelPurchaseReceipt interface

Use the `UnityChannelPurchaseReceipt` to further unpack a unified receipt into a Unity Channel receipt. See the example below:

```
var ucReceipt = JsonUtility.FromJson<UnityChannelPurchaseReceipt>(unifiedReceipt
```

Use this additional Unity Channel receipt data for informational purposes.

Key:	Type:	Description:
productID	string	The unique ID of the purchased Product
transactionID	string	The unique transaction ID
orderQueryToken	string	The order query token

Store identification at runtime

To check if the Xiaomi app store is the active store in your game at runtime, use the boolean value provided by the following code snippet:

```
var module = StandardPurchasingModule.Instance();
bool m_IsUnityChannelSelected =
    Application.platform == RuntimePlatform.Android &&
    module.androidStore == AndroidStore.XiaomiMiPay;
```

Implementing a Store

Leave feedback

Your store must implement the IStore interface, the methods of which are detailed in following sections.

```
using UnityEngine.Purchasing.Extension;

public class MyStore : IStore
{
    private IStoreCallback callback;
    public void Initialize (IStoreCallback callback)
    {
        this.callback = callback;
    }

    public void RetrieveProducts (System.Collections.ObjectModel.ReadOnlyCollect
    {
        // Fetch product information and invoke callback.OnProductsRetrieved();
    }

    public void Purchase (UnityEngine.Purchasing.ProductDefinition product, stri
    {
        // Start the purchase flow and call either callback.OnPurchaseSucceeded(
    }

    public void FinishTransaction (UnityEngine.Purchasing.ProductDefinition prod
    {
        // Perform transaction related housekeeping
    }
}
```

Initialization

Leave feedback

Your store's `Initialize` method is called by **Unity IAP** with an `IStoreCallback` that your store uses to communicate back to Unity IAP asynchronously.

```
void Initialize(IStoreCallback callback) {
    // Keep a reference to the callback for communicating with Unity IAP.
    this.callback = callback;
}
```

Retrieving products

Leave feedback

When your store's `RetrieveProducts` method is called it should fetch the latest product metadata and, optionally, ownership status for the current user.

When this process completes your store should call the `OnProductsRetrieved` method of the `IStoreCallback` supplied to your store upon initialisation, supplying a collection of `ProductDescription` that represent the items available for purchase.

Where products are owned by the user, your store may fill in the receipt and transaction ID fields of `ProductDescription`; **Unity IAP** will invoke the application's `ProcessPurchase` method for any transactions the application has not already processed.

Note that if the user is offline your store should retry until the user regains connectivity, taking care to avoid impacting game performance through aggressive polling.

Handling errors

If products cannot be retrieved due to an unrecoverable error, such as the developer making an error with their store configuration, you should call the `OnSetupFailed` method of the `IStoreCallback`, indicating the `InitializationFailureReason` responsible.

Handling purchases

Leave feedback

Your Store's Purchase method is called when the user opts to make a purchase. Your store should take the user through the checkout process and call either the `OnPurchaseSucceeded` or `OnPurchaseFailed` method of the `IStoreCallback`.

Your store should supply a receipt and unique transaction ID; if the application has not already processed a purchase with the supplied tranasaction ID, **Unity IAP** will invoke the application's `ProcessPurchase` method.

Finishing Transactions

When the application acknowledges that a transaction has been processed, or if the transaction has already been processed, Unity IAP invokes your store's FinishTransaction method.

Stores should use FinishTransaction to perform any housekeeping following a purchase, such as closing transactions or consuming consumable products.

Store Modules

Leave feedback

Store modules extend the `AbstractPurchasingModule` class, acting as factories **Unity IAP** can use to obtain an instance of your store along with any configuration and extensions.

Developers can supply multiple modules to Unity IAP, allowing them to use your custom store implementation alongside the default Unity-provided stores:

```
ConfigurationBuilder.Instance (MyCustomModule.Instance(), StandardPurchasingModu
```

Where two or more modules have implementations available for a given platform, precedence is given in order the modules were supplied to the `ConfigurationBuilder`; any implementation provided by `MyCustomModule` will be used in preference to `StandardPurchasingModule`.

Note that a module can support multiple stores; the `StandardPurchasingModule` handles all of Unity IAPs default store implementations.

Registering your store

Leave feedback

Call the `RegisterStore` method supplying a name for your store and your implementation, which must implement the IStore interface.

```
public override void Configure() {
    RegisterStore("GooglePlay", InstantiateMyStore());
}

private void InstantiateMyStore() {
    if (Application.platform == RuntimePlatform.Android) {
        return new MyAlternativeGooglePlayImplementation ();
    }
    return null;
}
```

The store name must match the name developers use when defining products for your store so **Unity IAP** uses the correct product identifiers when addressing your store.

Store Configuration

Leave feedback

Your store may require developers to supply additional configuration information during initialization, for which your module can register a configuration instance that implements the `IStoreConfiguration` interface:

```
var config = new MyConfiguration(); // Implements IStoreConfiguration
BindConfiguration<MyConfiguration>(new MyConfiguration());
```

When developers request an instance of your configuration type, **Unity IAP** first tries to cast your store implementation to the configuration type. Only if that cast fails will any instance bound via `BindConfiguration` will be used.

Store Extensions

Leave feedback

Your store may offer additional functionality that does not fit into the cross platform purchase flow, for example the ability to refresh app receipts on Apple's stores.

You should create an interface that defines the extended functionality, itself implementing the `IStoreExtension` interface:

```
/// <summary>
/// Functionality specific to my store.
/// </summary>
public interface IMyExtensions : IStoreExtension
{
    // Hypothetical method for a store that provides User IDs.
    String GetUserStoreId();
}
```

Applications request extended functionality via the `IExtensionProvider`. When they do so **Unity IAP** first tries to cast the active store implementation to the requested type.

If that cast fails, Unity IAP will provide any instance registered via a call your store module has provided via `RegisterExtension`, or null if no instance has been provided.

Modules should provide instances for the extension interfaces they define even when running on unsupported platforms, so as to avoid forcing application developers to use platform dependent compilation.

IAP Promo

Leave feedback

Overview

IAP Promo allows developers to easily promote micro-transactions. Use it to target different players with unique purchase opportunities at various points in the game.

Example IAP Promo showcase

Requirements

Your game must use Unity version 5.4+, and enable the following services:

Unity IAP SDK 1.17+
Unity Ads SDK 2.2+ (without mediation)

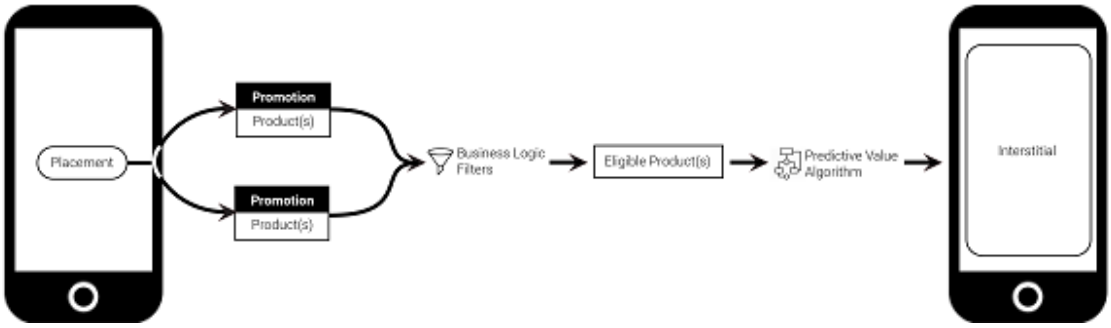
Note: Unity recommends using the latest Unity IAP SDK (1.2+) and Unity Ads SDK (2.3+) for access to the latest features. Using older versions of the SDKs may cause initialization errors.

IAP Promo compoments

There are three major **components** of an IAP Promo, all of which have their own articles:

Placements
Products
Promotions

Placements trigger fill requests, whereby the feature searches associated **Promotions** for targeted **Products** to show the player. Players can make in-app purchases directly from the resulting interstitial ad.



High-level overview of the IAP Promo fill request process

Maximizing revenue with machine learning

IAP Promo eliminates the need for traditional A/B testing methods. Its machine learning software leverages the aggregated data in Unity's network, looks for data patterns, then feeds them into a model to improve your game. This enables personalized offers, even for brand new players. This completely automated service works around the clock.

How it works

When a **Placement** makes a request, the system looks at all available **Products** that are eligible to display. It then applies business criteria filtering (for example, geographic or OS targeting), and uses data science to select the best Product for a user given the remaining options. The model considers many data points to inform its choice, such as device characteristics and player session behavior. We define the best offer as the Product that maximizes lifetime value (LTV).

Getting started

To leverage IAP Promo's machine learning feature, simply add multiple Products to a Promotion, and let Unity do the work.

Consider for example a **Promotion** that targets players when they complete a level:

When the player completes a level, the game calls an 'EndLevel' **Placement**. The Placement's fill request returns a list of three possible **Products**:
A $1.99 currency bundle with a 10% bonus
A $2.99 currency bundle with a 10% bonus
A $4.99 currency bundle with a 10% bonus
The system finds that the player meets targeting criteria for all three offers.
The algorithm determines which currency bundle best maximizes the player's **LTV**, and displays that offer.

We recommend supplying at least three price points per **Promotion** (low, medium, and high), with appropriately commensurate payouts. As with any data-driven model, the more data points available, the more effective it is. There is no limit to the number of **Products** you can add to a **Promotion**; you may even choose to provide a **Product** for each price tier in the game's app store.

2018–09–10 Page amended with editorial review

IAP Promo integration

Leave feedback

Overview

Important note: Your game must initialize **Unity IAP** before initializing Unity Ads for IAP Promo to work.

This integration guide covers four major steps:

Preparing your Project in the Unity Editor
Implementation
IAP
Ads
Configuring Promotions
Testing your integration

Preparing your Project in the Unity Editor

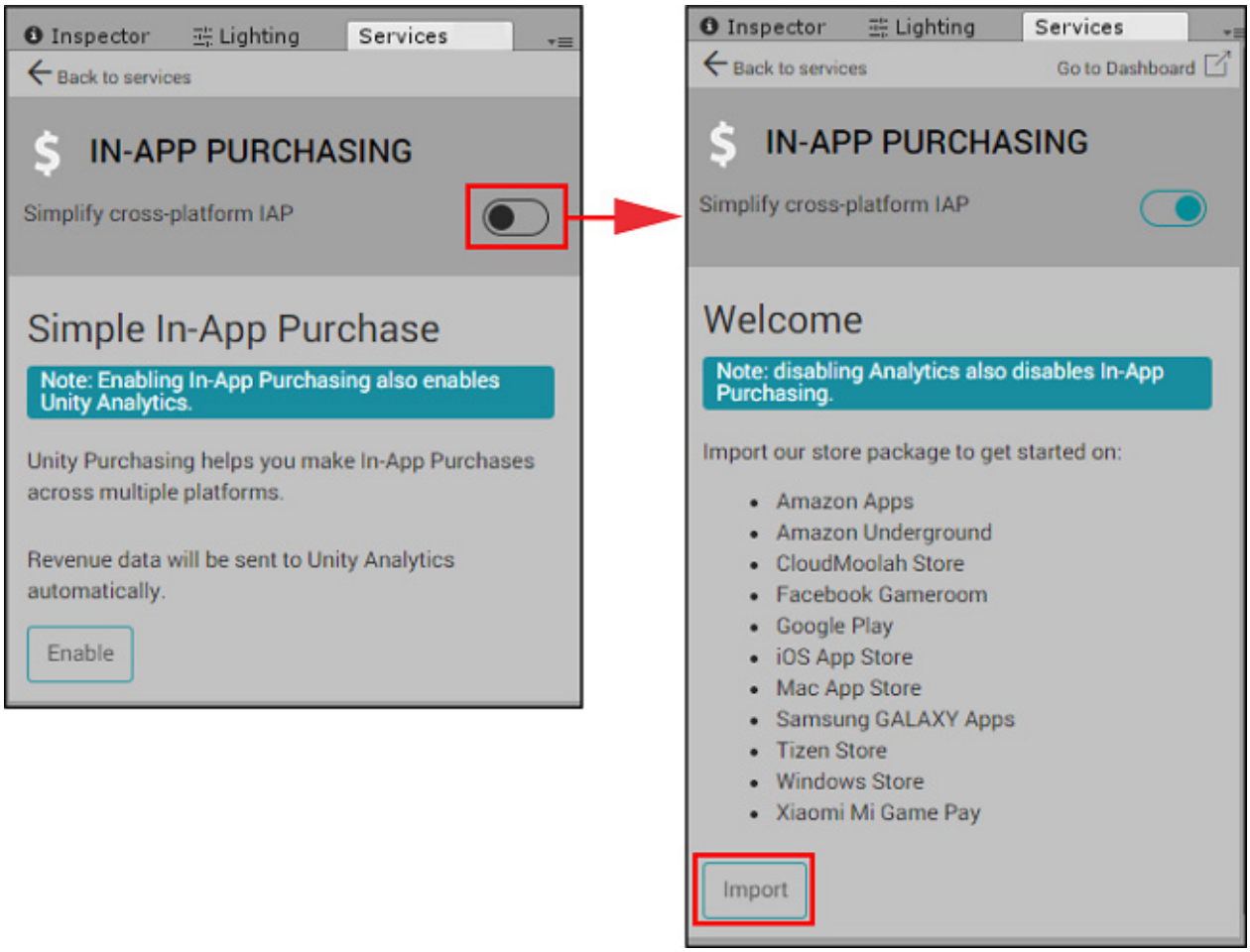
Setting up Unity Services

To use IAP Promo, you need to:

Configure your project for Unity Services.
Enable the Unity IAP SDK (1.2+) and Unity Ads SDK (2.3+) in your Project.

Setting up Unity IAP

IAP Promo requires a supported version of the Unity IAP SDK (1.17+). To acquire the latest IAP SDK, either enable **In-App Purchasing** in the Services window (**Window > Services**), or import it from the Asset store. If you're enabling it from the Services window, be sure to **Import** the Asset package when prompted.



Enabling Unity IAP in the Editor's Services window

See documentation on Setting up IAP for additional information.

Setting up Unity Ads

IAP Promo requires a supported version of the Unity Ads SDK (2.2+). Acquire the latest Ads SDK by importing it from the Asset store. This enables Unity Ads for your Project.

See Setting up Ads for Unity for additional information.

Implementation

With the required services set up, you can implement them in your game.

Implementing IAP

There are two options for initialization: codeless or scripting.

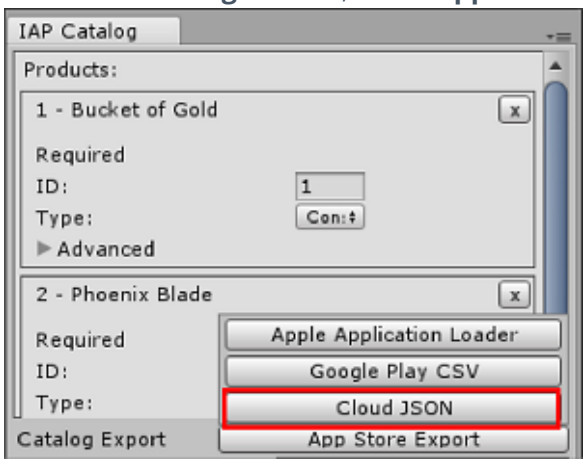
Using Codeless IAP

Codeless IAP handles initialization for you. If you use Codeless IAP initialization, you must call the Unity Ads initialization method elsewhere in your code.

To use Codeless IAP, populate a **Product Catalog**, then create an **IAP Listener** to fetch that catalog:

In the Editor, select **Window > UnityIAP > IAP Catalog** to open the **IAP Catalog** window. This window lists all of your previously configured Products. You must have at least one **Product** configured in your **Product Catalog**. For a complete walkthrough on setting up **Products**, see Codeless IAP.

In the **IAP Catalog** window, select **App Store Export > Cloud JSON** to export a local copy of the **Product Catalog**.



Create an **IAP Listener**. Select **Window > Unity IAP > Create IAP Listener**, and add it to the first scene of your game. The listener fetches your **Product Catalog** as soon as the game boots. This avoids errors where the game requests **Promotions** but a **Product** isn't ready because the codeless button hasn't appeared in the scene yet.

Using scripting

If you do not use Codeless IAP, you must initialize Unity IAP manually through a script. See the following code example:

```
using System.Collections;
using System.Collections.Generic;

using UnityEngine;
using UnityEngine.Events;
using UnityEngine.Purchasing;
using UnityEngine.UI;

public class UnityIAP : MonoBehaviour, IStoreListener
{
    private IStoreController controller;

    private const string product_coins = "100.gold.coins";
    private const string product_hat = "top_hat";
    private const string product_elite = "elite_status";
    private const string product_bundle = "gem_super_box";
    public int coin_count = 0;
    public int gems_count = 0;
    public bool hat_owned = false;
    public bool elite_member = false;

    private void Start()
    {
        Debug.Log("UnityIAP.Init()");

        StandardPurchasingModule module = StandardPurchasingModule.Instance();
```

```
        ProductCatalog catalog = ProductCatalog.LoadDefaultCatalog();
        ConfigurationBuilder builder = ConfigurationBuilder.Instance(module);
        IAPConfigurationHelper.PopulateConfigurationBuilder(ref builder, catalog);

        UnityPurchasing.Initialize(this, builder);
    }

    public void OnInitialized(IStoreController controller, IExtensionProvider exten
    {
        Debug.Log("UnityIAP.OnInitialized(...)");
        this.controller = controller;

        // Include any additional initialization logic as needed
    }

    public void OnInitializeFailed(InitializationFailureReason error)
    {
        Debug.Log("UnityIAP.OnInitializeFailed(" + error + ")");

        // Include any additional initialization failure logic as needed
    }

    public void Buy(string productId)
    {
        Debug.Log("UnityIAP.BuyClicked(" + productId + ")");
        this.controller.InitiatePurchase(productId);
    }

    public void OnPurchaseFailed(Product item, PurchaseFailureReason r)
    {
        Debug.Log("UnityIAP.OnPurchaseFailed(" + item + ", " + r + ")");
    }

    public PurchaseProcessingResult ProcessPurchase(PurchaseEventArgs e)
    {
        string purchasedItem = e.purchasedProduct.definition.id;

        switch (purchasedItem)
        {
            case product_coins:
                Debug.Log("IAPLog: Congratualtions you are richer!");
                coin_count += 100;
                Debug.Log("IAPLog: Coin count: " + coin_count);
                break;

            case product_hat:
                hat_owned = true;
                Button topHatButton = GameObject.Find("buyTopHat").GetComponent<But
```

```
                topHatButton.interactable = false;
                topHatButton.GetComponentInChildren<Text>().text = "That hat is dasl
                Debug.Log("IAPLog: Hat owned: " + hat_owned);
                break;

            case product_elite:
                elite_member = true;
                Button eliteButton = GameObject.Find("buyElite").GetComponent<Butto
                eliteButton.interactable = false;
                eliteButton.GetComponentInChildren<Text>().text = "Welcome to Elite
                Debug.Log("IAPLog: Elite member: " + elite_member);
                break;
            case product_bundle:
                gems_count += 5000;
                break;
        }

        return PurchaseProcessingResult.Complete;
    }
}
```

Implementing Unity Ads

You must also initialize Unity Ads, whether or not you use the Codeless or manual IAP initialization method. The following code sample illustrates an initialization method to invoke:

```
using System.Collections;
using System.Collections.Generic;

using UnityEngine;
using UnityEngine.Advertisements;
using UnityEngine.Events;
using UnityEngine.Purchasing;
using UnityEngine.UI;

public class AdsManager : MonoBehaviour
{
    #if UNITY_IOS
        private string gameId = "0000000"; // Your iOS game ID here
    #elif UNITY_ANDROID
        private string gameId = "9999999"; // Your Android game ID here
    #else
        private string gameId = "0123456"; // Prevents Editor Errors
    #endif
```

```
    public void Init()
    {
        Debug.Log("UnityAds.Init()");
        if (!Advertisement.isSupported || Advertisement.isInitialized)
        {
            Debug.Log("Could not initialize ads");
            return;
        }

        Debug.Log("Initializing Unity Ads with game ID: " + gameId);
        Advertisement.Initialize(gameId, false);
    }

    public void ShowAdUnit()
    {
        Debug.Log("Unity Ads Log: Ad shown");
        Advertisement.Show("testAdButton"); // Ad Placement ID for ad here
    }

    public void ShowPromo()
    {
        Debug.Log("Unity Ads Log: Promo Shown");
        Advertisement.Show("testPromoButton"); // Ad Placement ID for Promo here
    }
}
```

Configuring Promotions on the Developer Dashboard

Navigate to the IAP Promo section of the Developer Dashboard to configure your IAP Promo offers:

Use Placements to control when and how your **Promotions** display in-game.
Use the Products interface to import your **Product Catalog** and manage each Product's creative **assets**.
Define the parameters of your Promotions, such as when to run them, which **Placements** and **Products** they include, and which users they target.

Testing your integration

Call your IAP Promo content by implementing the following example code:

```
public void ShowPromo()
{
    Advertisement.Show (placementID);
}
```

Press **Play** in the Editor to check that a test ad appears when the **Placement** makes its request. To see real promotional creative assets, you must build the game to a device in production mode.

2018–09–10 Page amended with editorial review

IAP Promo Placements

Leave feedback

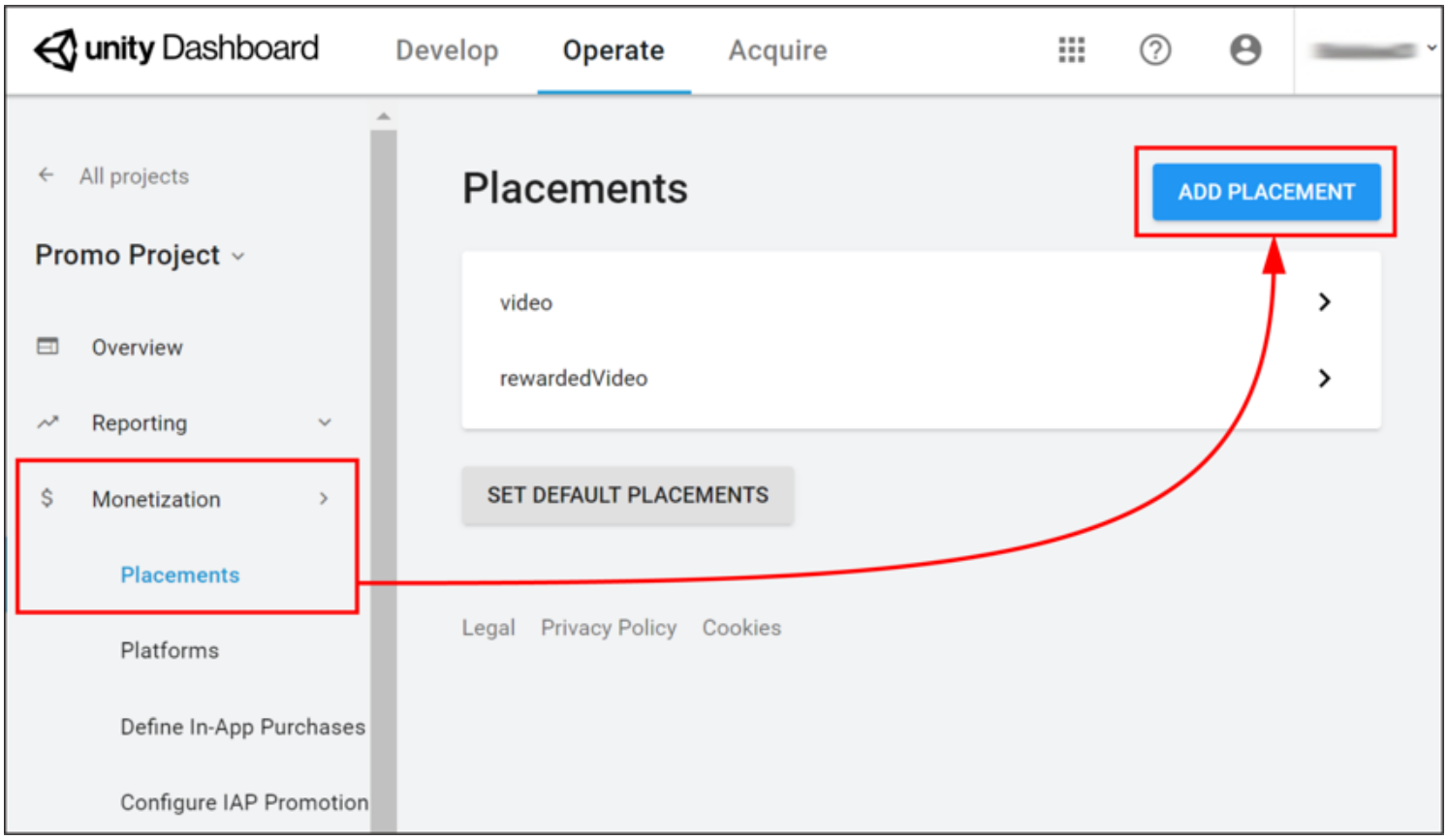
Use **Placements** to control when and how your **Promotions** display in-game. You create **Placements** on the Developer Dashboard, then call them via code in the Unity Editor, similar to normal Unity Ads Placements.

Creating Placements

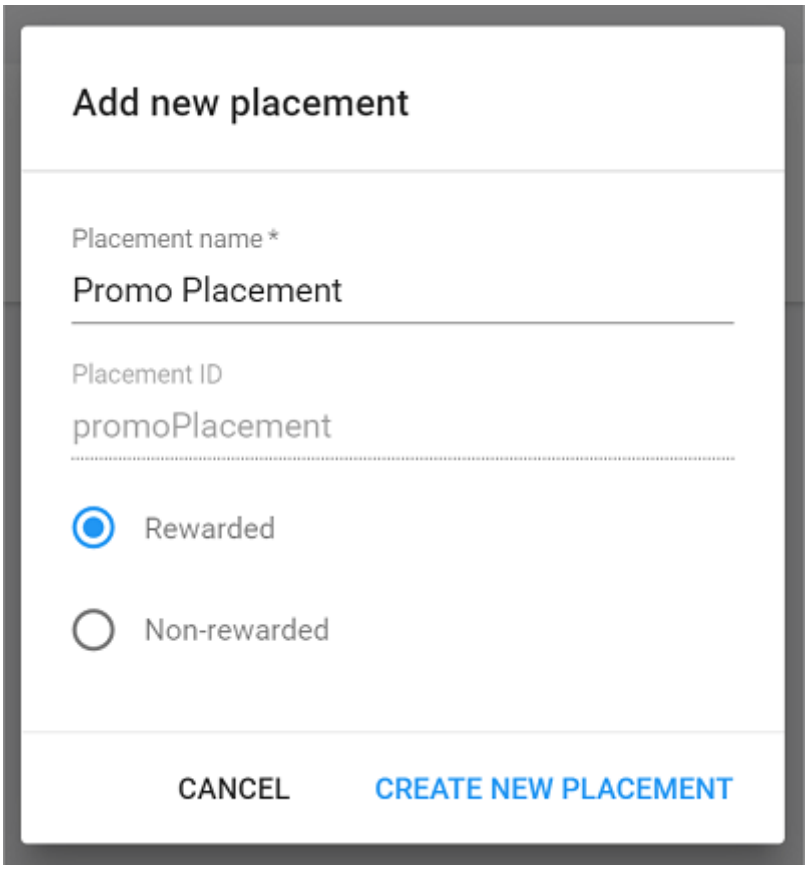
Create Promo **Placements** on the Operate page of the Developer Dashboard. Your active Project appears in the top-left corner. To create a Promo **Placement**:

Select **Monetization** > **Placements** from the left navigation bar.

Click the **ADD PLACEMENT** button in the top-right corner.



Enter a **Placement Name**, then select whether the Placement is **Rewarded** or **Non-rewarded** (for more information on rewarded and non-rewarded ad Placements, see documentation on monetization best practices).

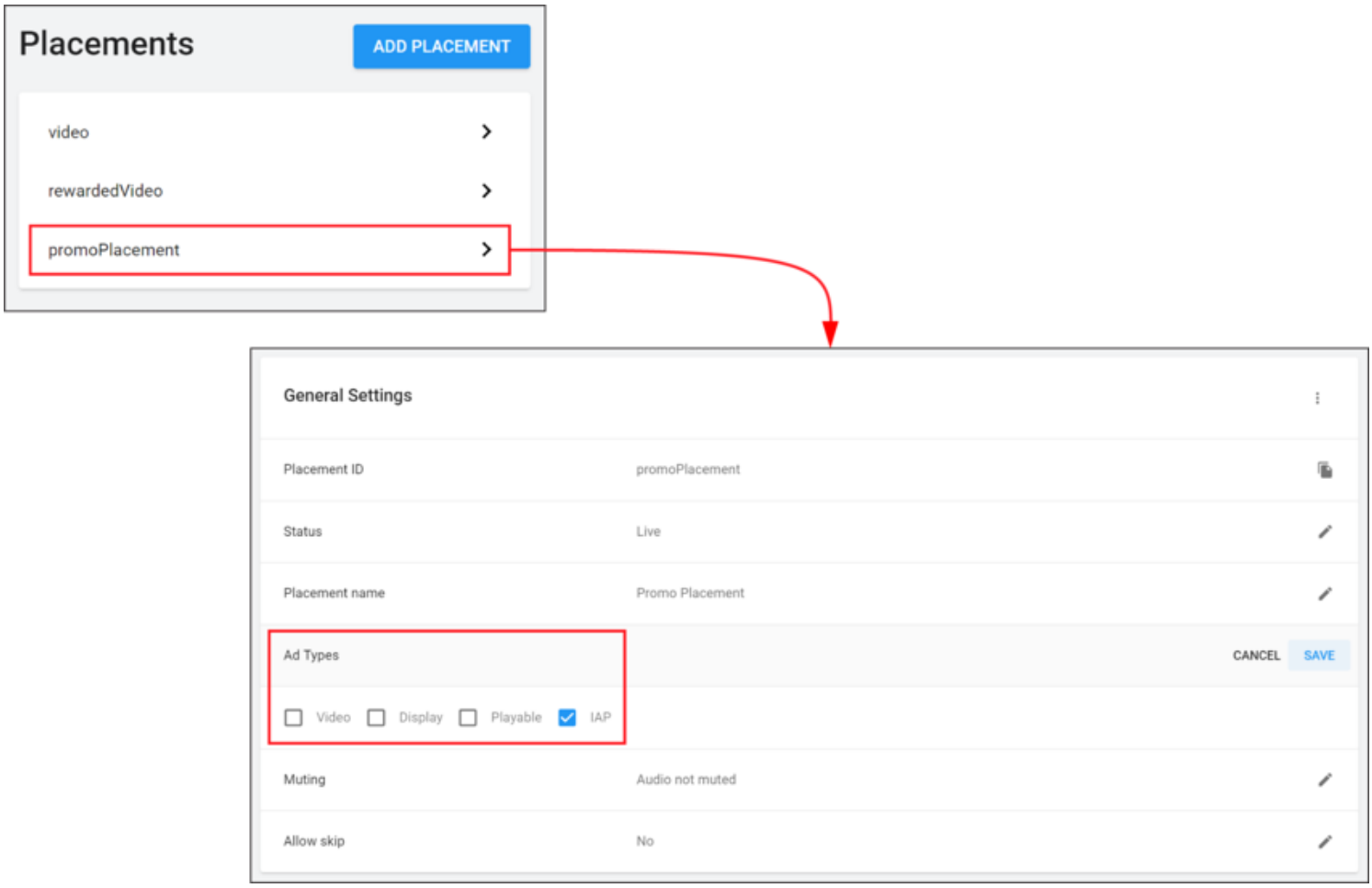


Note: The dashboard auto-populates **Placement IDs** based on the Placement's name. You cannot edit **Placement IDs** after the **Placement** is created, even if you updated the name. Use them to call **Placements** in your code using the Unity Ads API (see Unity Ads documentation on Placements and filters for more information).

Select **CREATE NEW PLACEMENT** to save the new Placement and return to your Project's Placements list.

Select the Placement you created for IAP Promo to open its **General Settings** menu.

Expand the **Ad Types** setting, then check **IAP** and click the **SAVE** button.



Your Placement is now configured for use with IAP Promotions.

2018–09–10 Page amended with editorial review

IAP Promo Product Catalogs

Leave feedback

Unity IAP provides quick and easy tools to create and export **Product Catalogs**. Use the Unity Editor's **IAP Catalog** window to populate **Product** metadata, then export the catalog to the Developer Dashboard to configure each Product's creative **assets**. Note that you must use the IAP Catalog function to export Products to the dashboard.

Creating a Product Catalog

Adding Products in the IAP Catalog window

In the Editor, select **Window > UnityIAP > IAP Catalog** to open the **IAP Catalog** window. This GUI lists all of your previously configured **Products**, and allows you to define the following attributes for each:

Name
Product ID
Type (IAP Promo now supports both **Consumable** and **Non-consumable** Product Types)
Price (Unity recommends specifying price points for Google and Apple stores independently if your app supports both)
Note: A **Product Catalog** must have at least one **Product** configured to export. For a complete walkthrough on setting up **Products**, see documentation on Codeless IAP.

You may already use a builder script to populate your **Product Catalog** for **Unity IAP**. For example:

```
var builder = ConfigurationBuilder.Instance(StandardPurchasingModule.Instance())
builder.AddProduct(product_coins, ProductType.Consumable);
builder.AddProduct(product_hat, ProductType.NonConsumable);
builder.AddProduct(product_elite, ProductType.Subscription);
```

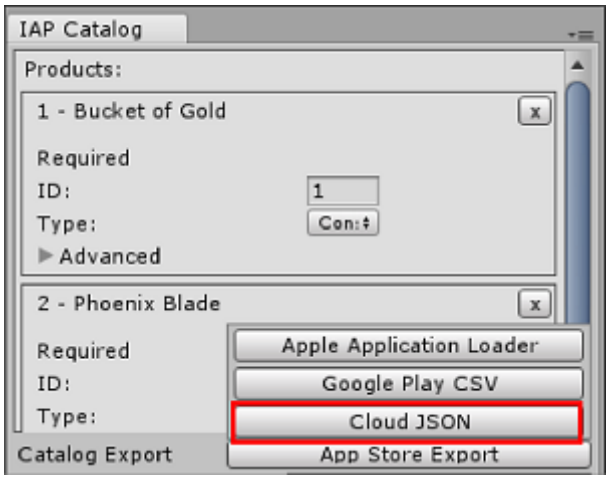
Because IAP Promo only supports JSON exports from the Editor **IAP Catalog** GUI, we recommend modifying your approach as follows:

Copy-paste your **Products** into the **IAP Catalog** GUI.
Remove or comment out existing `builder.AddProduct()` code from your purchasing script.
Add the following code before initializing IAP to fetch your Product list from the **IAP Catalog** GUI:

```
StandardPurchasingModule module     = StandardPurchasingModule.Instance();
ProductCatalog catalog = ProductCatalog.LoadDefaultCatalog();
ConfigurationBuilder builder = ConfigurationBuilder.Instance(module);
IAPConfigurationHelper.PopulateConfigurationBuilder(ref builder, catalog);

UnityPurchasing.Initialize(this, builder);
```

Exporting a Product Catalog

In the **IAP Catalog** window, select **App Store Export > Cloud JSON** to export a local copy of the **Product Catalog**.



Exporting a Product Catalog to JSON

Configuring creative assets in the Developer Dashboard

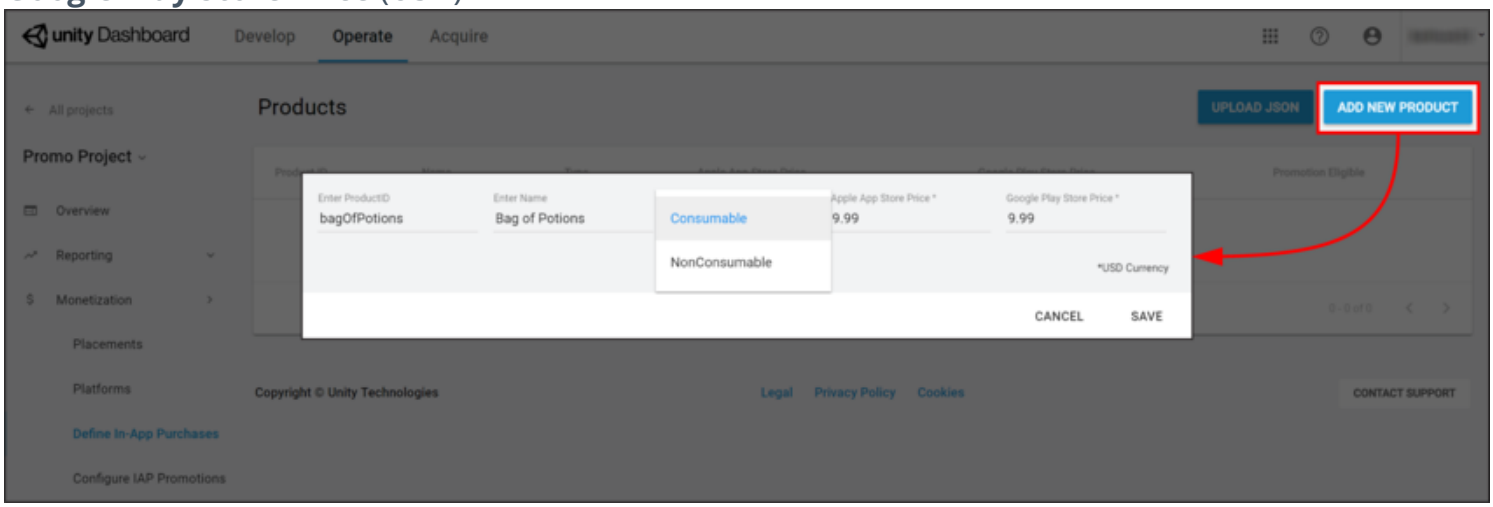
Manage your Product Catalog on the Operate page of the Developer Dashboard. Your active Project appears in the top-left corner. To manage Products, select **Monetization > Define In-App Purchases** from the left navigation bar.

You can populate your Products manually, or bulk-populate them by importing a JSON file.

Creating a Product manually

Click the **ADD NEW PRODUCT** button in the top-right corner to view the Product configuration window, then enter the following details:

Product ID (for reference in your IAP code)
Name
Consumable or **Non-Consumable**
Apple App Store Price (USD)
Google Play Store Price (USD)



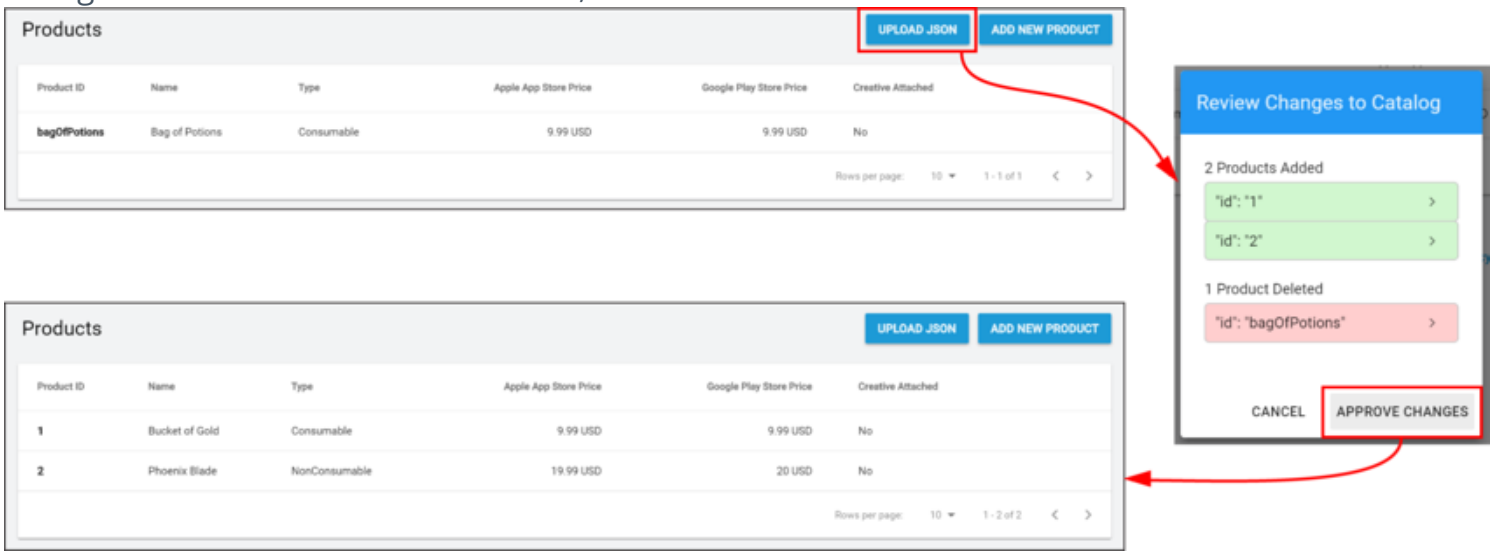
Adding a new IAP Product manually on the Developer Dashboard

Click the **SAVE** button to store the Product in your catalog.

Importing a Product Catalog

To set up a new **Product Catalog** or update existing **Products** in the catalog:

Select **UPLOAD JSON**.
Select the JSON file you exported from the Unity Editor. A diff log then prompts you to approve changes. Check the contents of the file, then select **APPROVE CHANGES**.



Importing your IAP Product Catalog to the Developer Dashboard

The imported **Products** appear in the **Product Catalog** list. However, in order to be eligible for **Promotions**, each **Product** must have an associated creative asset (otherwise there is nothing for the **Placement** to display).

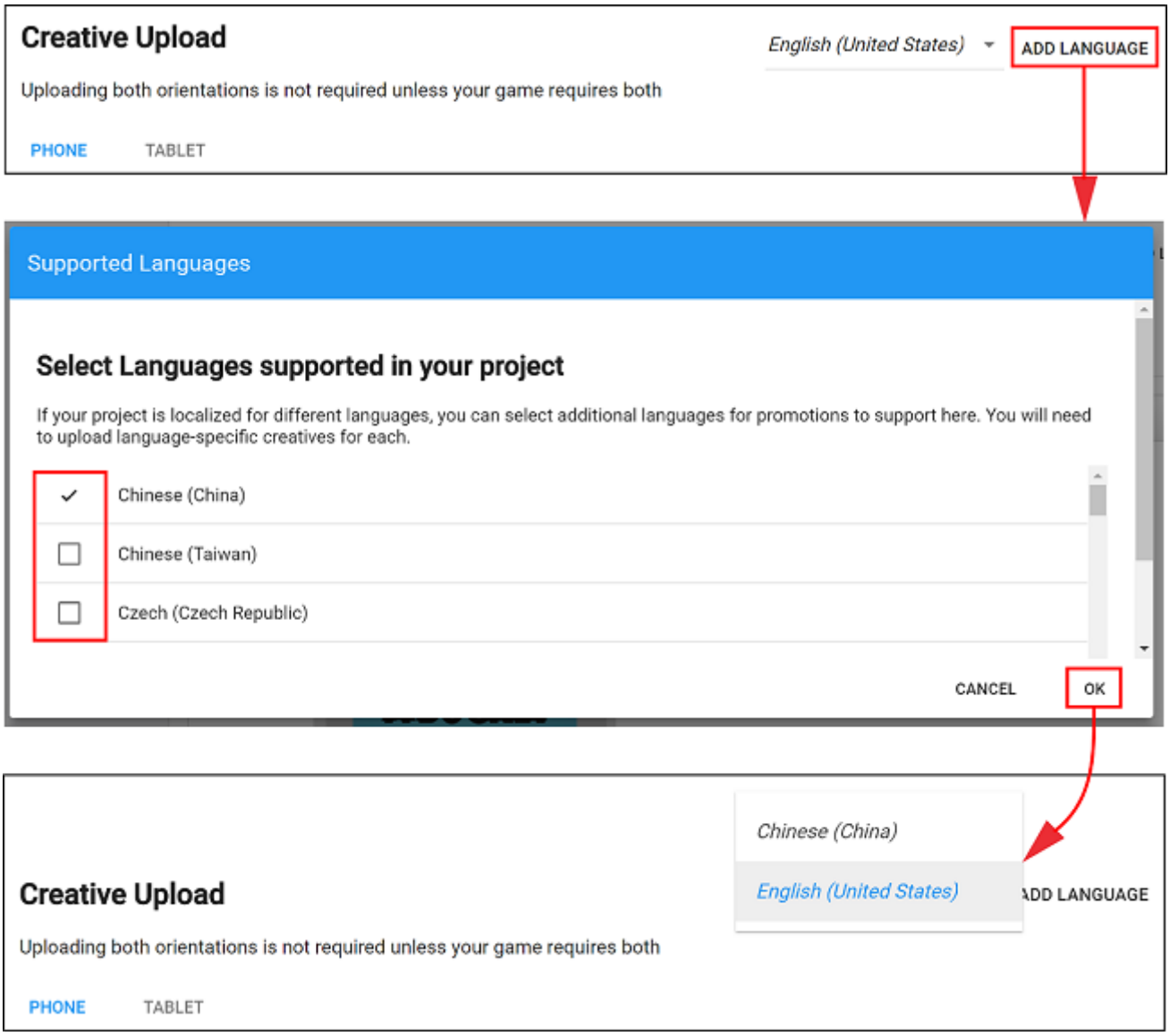
Assigning creative assets

To add creative assets, follow these steps:

Select a **Product** from the **Product Catalog** list, then scroll down to the **Creative Uploads** section.

Note: While each Product's **Product ID**, **Name**, **Price**, and **Type** appear in the dashboard, you can only edit these attributes from the Editor's **IAP Catalog** window.

To specify unique creative assets for different regions, select **ADD LANGUAGE**, then check the languages to include.



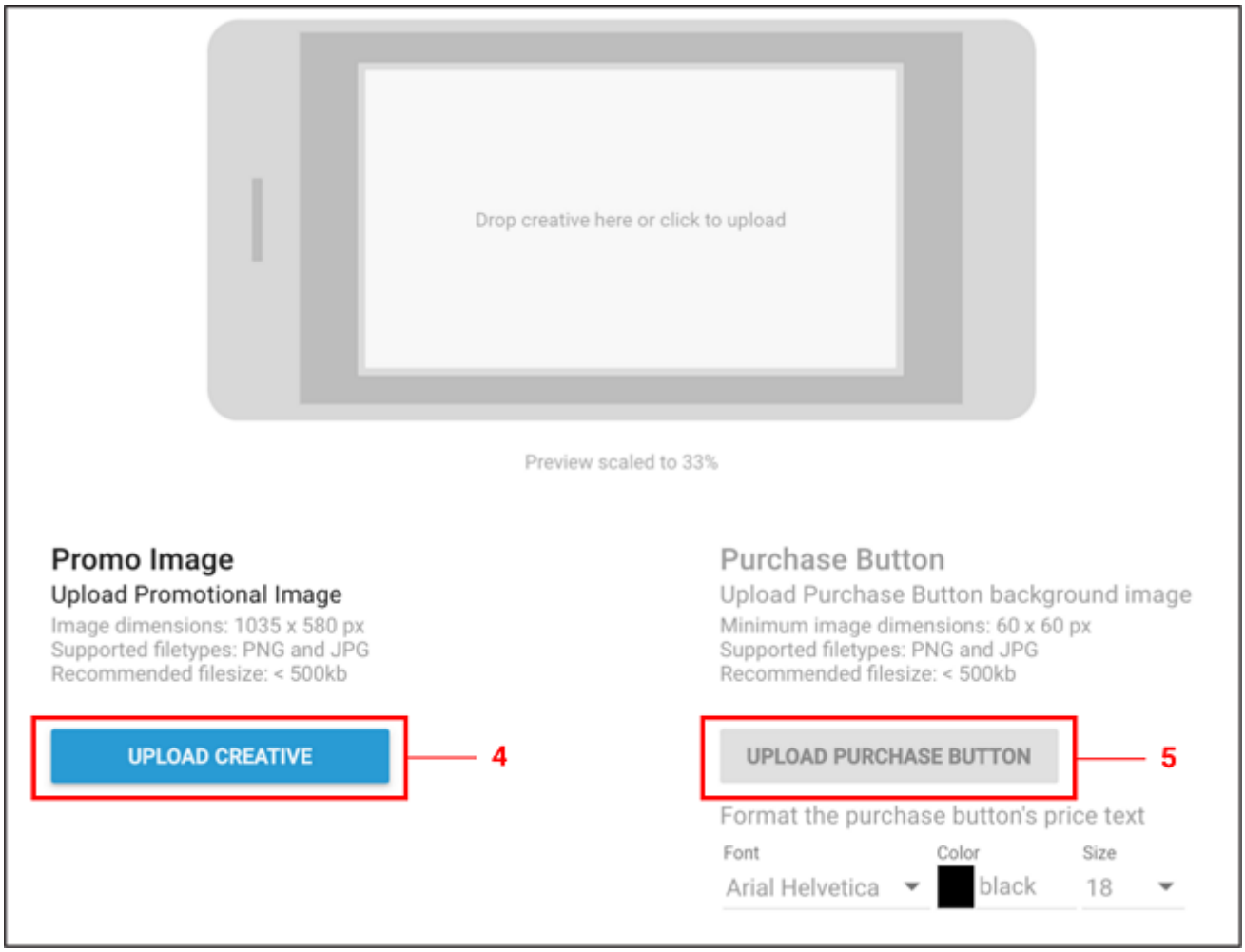
Note: The drop-down indicates the current language you are editing. If an included language has no creative assets, it defaults to using its **English (United States)** creative assets.

Upload **Portrait** or **Landscape** creative assets for phone and tablet devices, with the following guidelines:

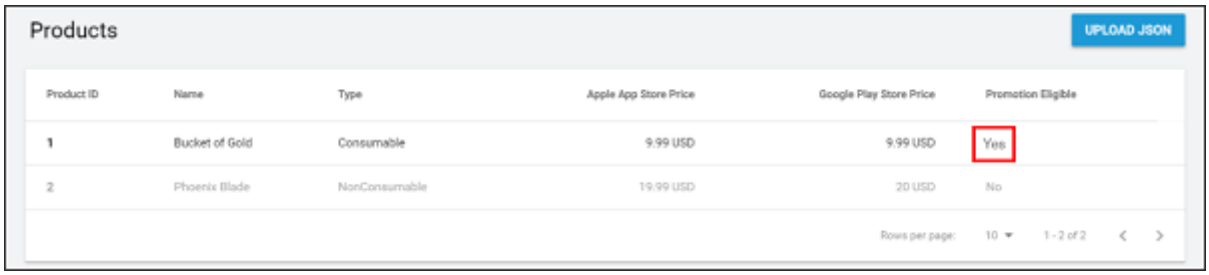
PNG or JPG file format.
Each **Product** requires a creative asset for at least one configuration to be eligible for **Promotions**.
For a better consumer experience, provide optimized creative assets for each device and orientation.
Phone creative asset dimensions are 580 x 1035 **pixels** for portrait orientation, and 1035 x 580 **pixels** for landscape orientation.
Tablet creative asset dimensions are 900 x 1200 pixels for portrait orientation, and 1200 x 900 pixels for landscape orientation.
Select **UPLOAD CREATIVE** to upload a creative asset. A preview appears on the left.

Select **UPLOAD PURCHASE BUTTON** to place an interactive button on the creative asset, with the following guidelines:

You must upload a purchase button to proceed.
Minimum file size for the button asset is 60 x 60 pixels.
You can click and drag the purchase button on the preview window to place it.
You can format the font, color, and size of the button text.



Select **SAVE** to proceed. Upon returning to the **Product Catalog** page, your **Product** is now **Promotion Eligible**.



Note: You must export and submit your **Product Catalog** to the appropriate Google Play or iTunes store for the purchase flow to work. Please see the **Export** section in the documentation on Codeless IAP.

2018–09–10 Page amended with editorial review

Promotions

Leave feedback

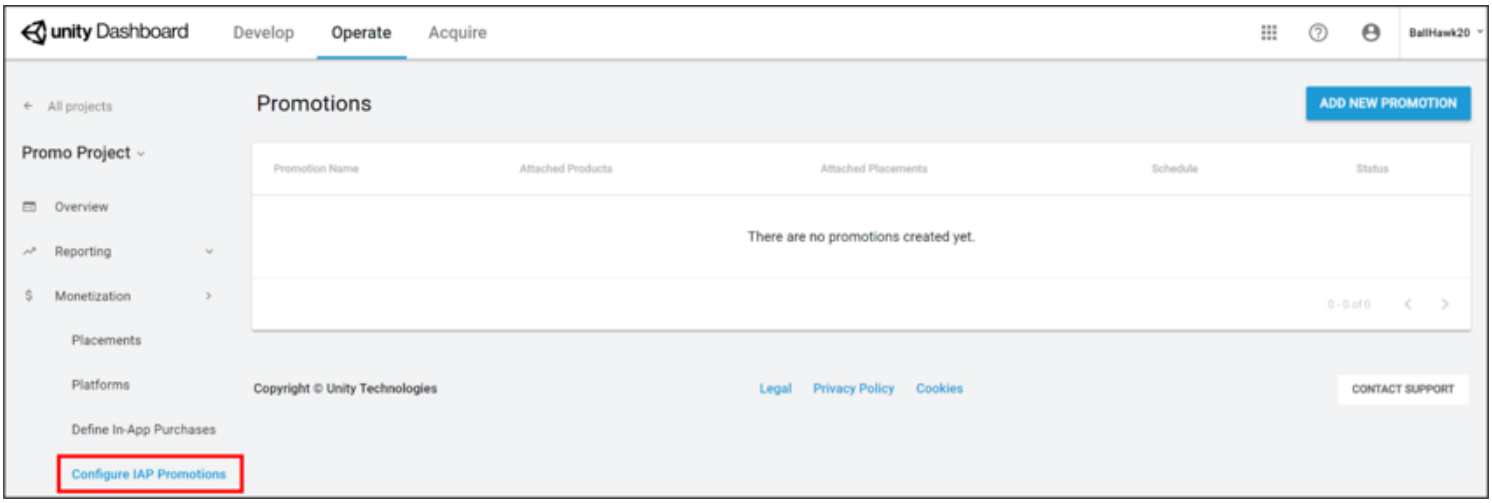
IAP **Promotions** work in confluence with **Placements** to target players with precise purchase opportunities tailored to their unique situations. There are three **components** of a **Promotion**:

Included **Products** (see documentation on IAP Promo Product Catalogs)
Associated **Placements** (see documentation on IAP Promo Placements)
Business logic (detailed below)
Create and manage **Promotions** exclusively from the Developer Dashboard.

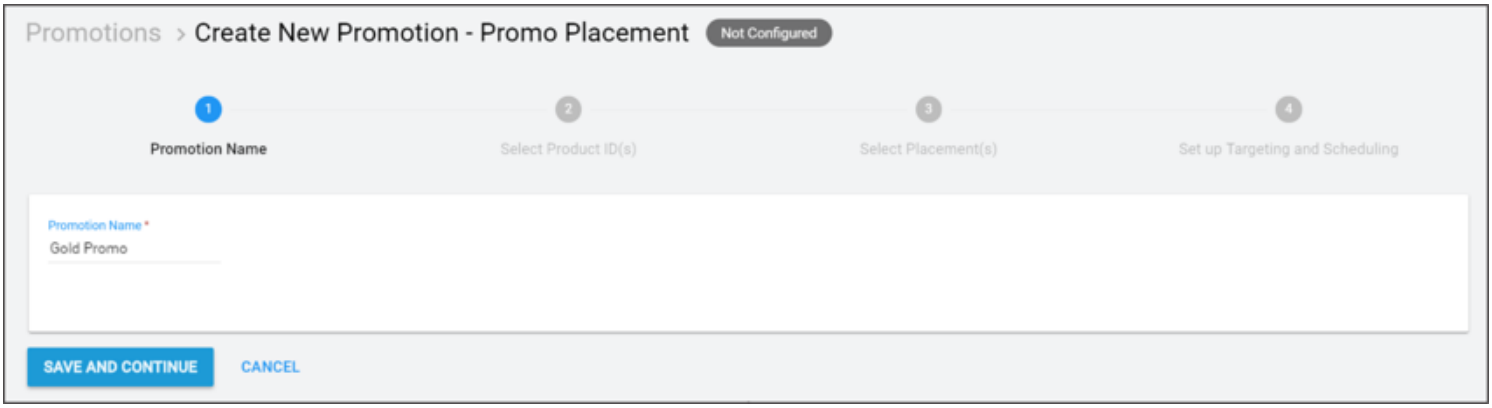
Creating Promotions

Create IAP **Promotions** from the Operate page of the Developer Dashboard. Your active Project appears in the top-left corner. To create a **Promotion**:

Select **Monetization > Configure IAP Promotions** from the left navigation bar. Any existing **Promotions** appear here.
Click the **ADD NEW PROMOTION** button in the top-right corner.



Enter a **Promotion Name**, then select **SAVE AND CONTINUE** to configure it.



Configuring Promotions

At any time during the configuration process, you can use the creation **timeline** to select the step you wish to configure.

Including Products

Click **2 Select Product ID(s)** on the creation timeline to include **Products** for your **Promotion**. Your Project's Product Catalog appears here. To include a **Product**, it must have an associated creative **asset**. Select the eligible **Product**(s) you wish to include, then click **SAVE AND CONTINUE** to confirm your selections.

Selecting Products for inclusion in your Promotion on the Developer Dashboard

For more information on configuring Products, see the IAP Promo Products documentation.

Note: **Placement** requests return a list of all potential **Products** to show. Selected **Products** that lack creative assets are ineligible for promotion, and get filtered out. See the **Machine learning** section of the IAP Promo overview for more information.

Including Placements

Click **3 Select Placement(s)** on the creation timeline to choose which **Placements** can show your **Promotion**. Your Project's **Placements** appear here. To include a **Placement**, it must be configured to show IAP. Select the eligible **Placement**(s) you wish to include, then click **SAVE AND CONTINUE** to confirm your selections.

Selecting Placements for inclusion in your Promotion on the Developer Dashboard

Note: Multiple **Placements** can target the same **Promotion**, and vice versa.

Defining business logic

Click **4 Set up Targeting and Scheduling** on the creation timeline to define the logic your **Promotion** uses to target customers.

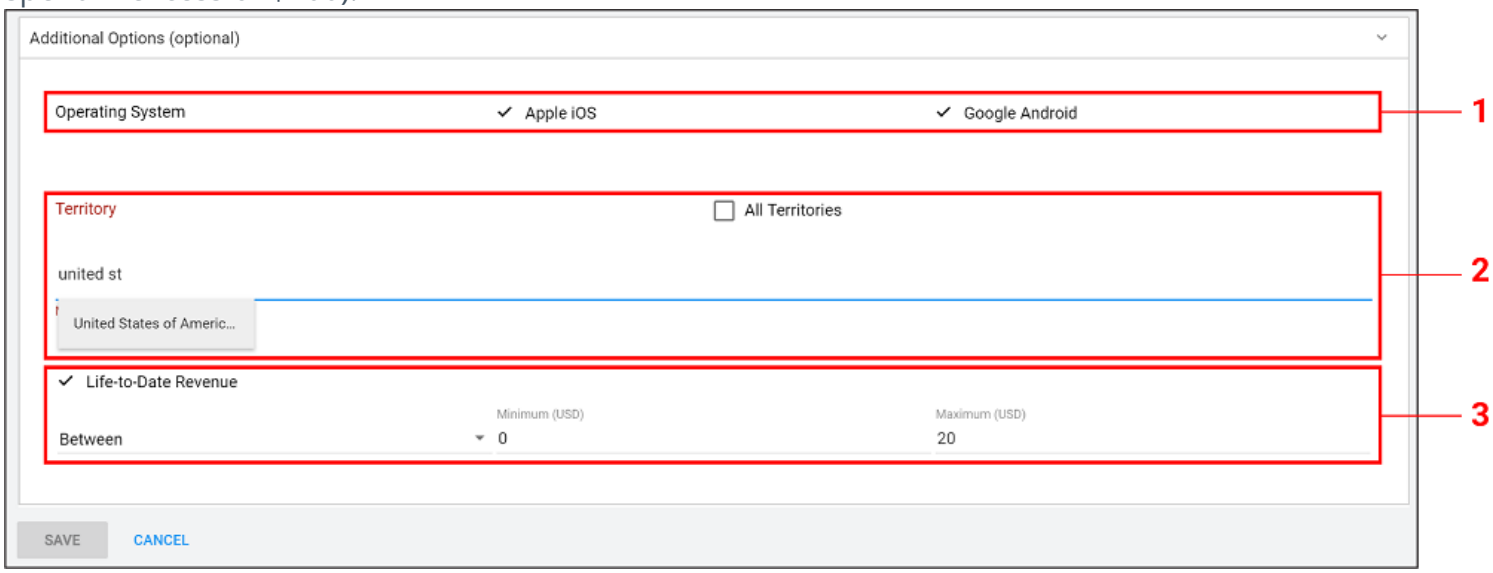
Specify an active period for the **Promotion** to run. Check **Begin Immediately** and **No End Date** for an indefinite duration, or select the calendar icons to specify a start and end date.

Specifying an active period for your Promotion on the Developer Dashboard

Additional Options

Expand the **Additional Options** menu for more targeted promotional strategies.

Operating System filters targeted players by **iOS** or Android devices.
Territory filters targeted players by country or region.
Life-to-Date Revenue filters targeted players by the amount they have spent on in-app purchases for their life-to-date in the game (for example, a **Promotion** can target non-payers, or players who have spent in excess of $100).



Selecting targeted business criteria for Promotions on the Developer Dashboard

Select **SAVE** to finish configuring the **Promotion**.

A successfully activated Promotion

You can view each Promotion's number of **Attached Products**, **Attached Placements**, **Schedule**, and **Status** from the **Configure IAP Promotions** page. Edit a **Promotion** at any time by clicking on it.

A successfully saved Promotion

2018–09–10 Page amended with editorial review

Unity Collaborate

Leave feedback

Unity Collaborate is part of Unity Teams. Unity Teams enables small teams to save, share, and sync a Unity Project in a cloud-hosted environment. Using Collaborate allows your entire team to contribute to a Project, regardless of location or role. For more information on Unity Teams, see the Teams page on the Unity3d website.

Collaborate allows you to publish your projects to the cloud for storage. Collaborate maintains a history of the versions of the Project that publish, allowing you to restore individual files or your entire Project to an earlier state.

You can add team members to your Projects; allowing you and team members to work together. Collaborate continuously monitors changes that are made by each team member and displays a badge on the files that they have changed, but not published yet.

You can view the changes. You can then choose to revert the changes on your file or publish them and handle any merge conflicts that might occur.

Collaborate is designed to support multi-disciplinary teams (that is, teams with a diverse mix of developers, artists, audio experts and other specialists) that work together on projects. Collaborate is in full release starting with Unity 2017.1. Collaborate is in beta for versions of Unity prior to 2017.1.

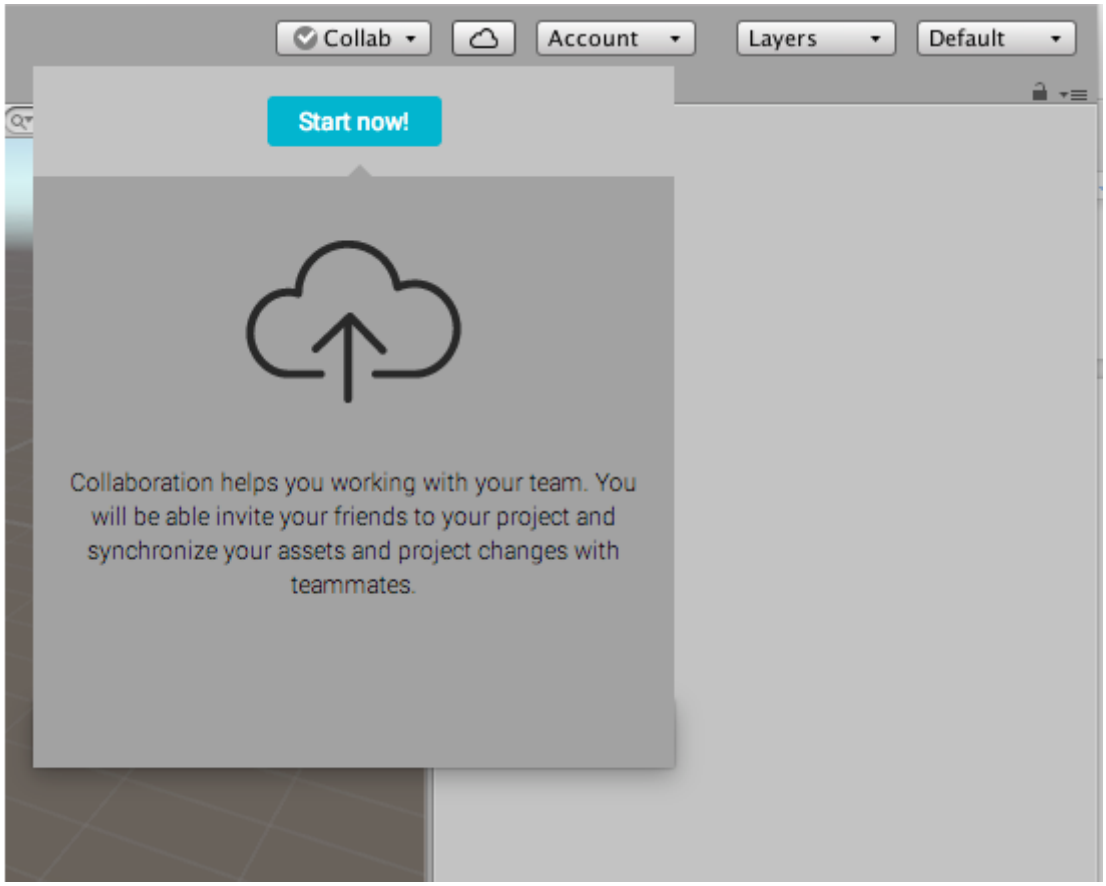
To enable Collaborate, open the **Services** window in the Unity Editor (menu: **Window** > **General** > **Services**). To launch Collaborate while you're working, click the **Collab** button in the Unity Editor.

Setting up Unity Collaborate

Leave feedback

Typically, Unity Services are enabled by default. You can specify settings during Project creation to disable Unity Services. If Unity Services are disabled, you must re-enable them before you can use Collaborate. For more information, see Setting up your Project for Unity Services.

To enable Collaboration in your Unity Project, on the Unity Editor **toolbar** click the **Collab** button and then click **Start Now**.

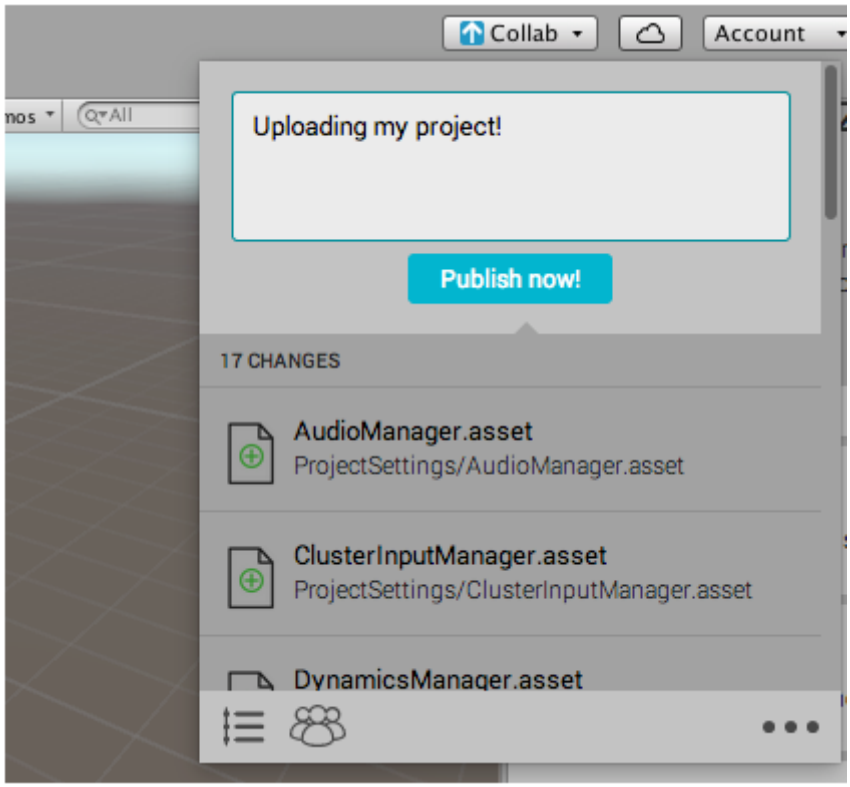


Start now dialog

If you enable Collaborate on a newly created Project, your Project is automatically published to the cloud.

If you are adding Collaborate to an existing Project, you might need to manually publish your Project.

To publish your changes and share them with your teammates, click the **Collab** button and then click **Publish now**.



Publish dialog

When you make changes to your Project, Collaborate indicates that you have unpublished changes by displaying an up arrow on the **Collab** button.

Up arrow

Adding team members to your Unity Project

Leave feedback

Unity Collaborate is part of Unity Teams. For each person you want to collaborate with, you must assign a Unity Teams seat for them in your organization.

Unity Teams is available in two versions:

Basic, which gives you 3 free seats.

Advanced, which gives you 3 seats after which you can purchase additional seats.

For more information on Unity Teams features, see the Unity Teams homepage.

This page describes how to do the following:

Enable Collaborate for a team member
Allow collaboration on all Projects in your organization
Allow collaboration on individual Projects
Manage team members in your organization
Manage Unity Teams member access

Enable Collaborate for a team member

There are two ways to enable collaboration with your team members. You can:

Allow collaboration on all Projects in your organization

Allow collaboration on individual Projects

To add team members for either of these scenarios, you must be a Manager or Owner of the organization that the Projects are part of.

To learn how to verify your role within an organization, see *Managing Your Organization* , below.

Allow collaboration on all Projects in your organization

To give a team member access to all of the Projects in your organization that have Collaborate is enabled:

Make them an Owner or Manager of the organization. To learn how to do this, see *Manage Your Organization* , below.

Give them access to Unity Teams. To learn how to do this, see *Managing Collaborators in your organization* , below.

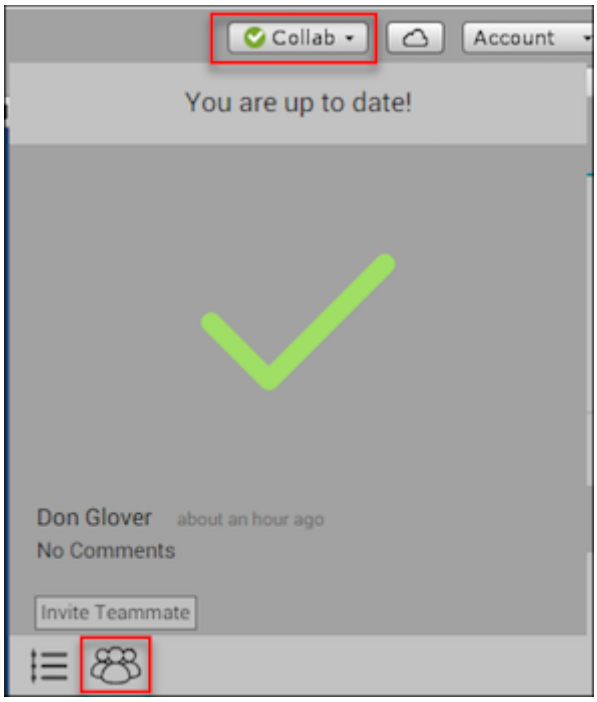
For more information on enabling collaboration, see documentation on Setting up Unity Collaborate.

Allow collaboration on individual Projects

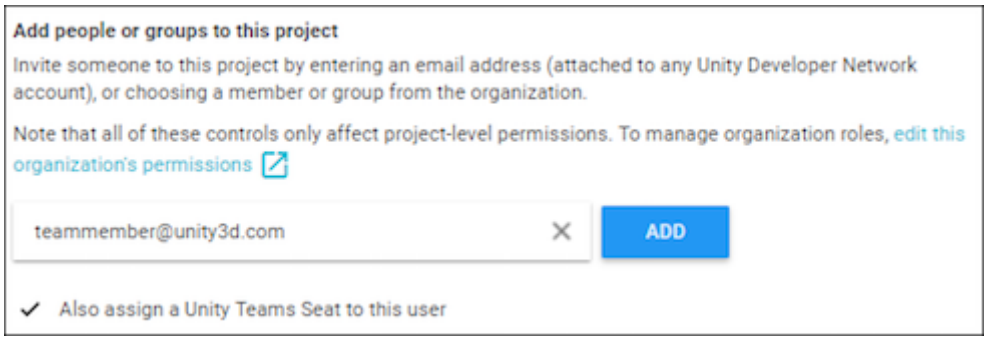
To give team members access to specific Projects:

On the Unity **toolbar**, click the **Collab** button.

In the Collaborate panel, click the **Invite Teammate** icon.



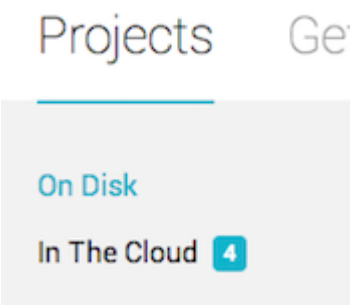
When you click the **Invite Teammate** icon, Unity launches the **Users** section of the Unity Developer Dashboard in your browser. From the **Users** section, you can invite anyone with a Unity ID to join your Project:



Note: You must tick **Also assign a Unity Teams Seat to this user** to enable the user to use Collaborate on your Project.

To add the user as a guest in your organization, click **ADD**. This assigns a Unity Teams seat to the user.

When your team member has access to the Project, they can see the Project in their Launcher (**Projects** > **In The Cloud**) and they can download the Project.



The Unity Launcher, with four projects available from cloud storage

Manage team members in your organization

Manage team members in the **Organizations** page of the Unity ID website. To manage team members:

Sign into Unity ID.

On the left-hand side of the page, click **Organizations**.

Select your organization.

On the left-hand side, select **Members & Groups.**

From this view you can see the members of your organization, as well as the role of each member. Users can have one of four roles: Owner, Manager, User, or Guest.

If you are an organization Owner, you can change the roles of all members. Additionally, you can add and remove members of all roles: Owner, Manager, User, and Guest.

If you are an organization Manager, you can change the role of Manager, User, and Guest members. Additionally, you can add and remove Manager, User, and Guest members.

For more information, see Unity's Knowledgebase article What can Owners, Managers and Users do in my Unity Organization?.

Manage Unity Teams member access

To access Collaborate and **Cloud Build** functionality, your team members must have access to Unity Teams. To check your team members' access:

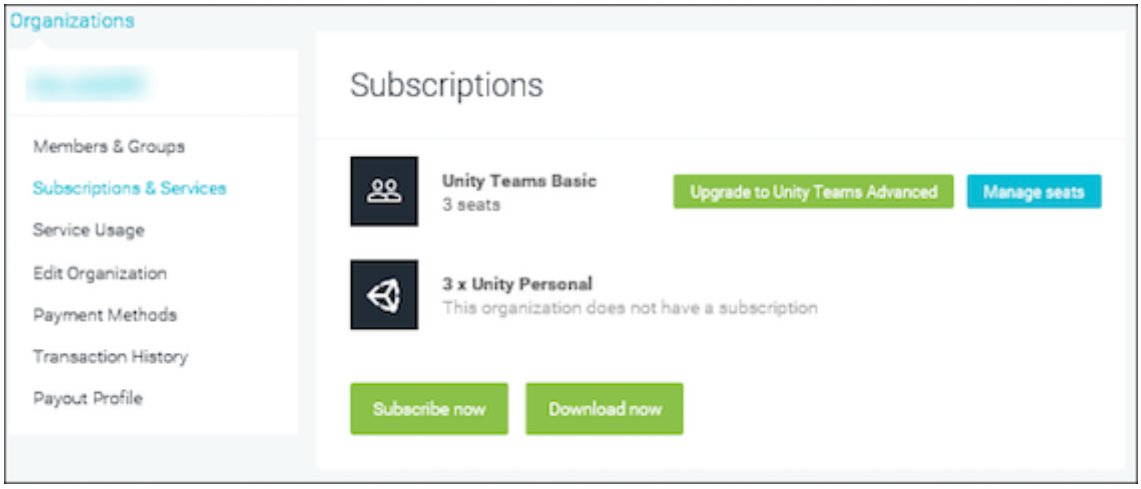
Sign into Unity ID.

On the left-hand side of the page, click **Organizations**.

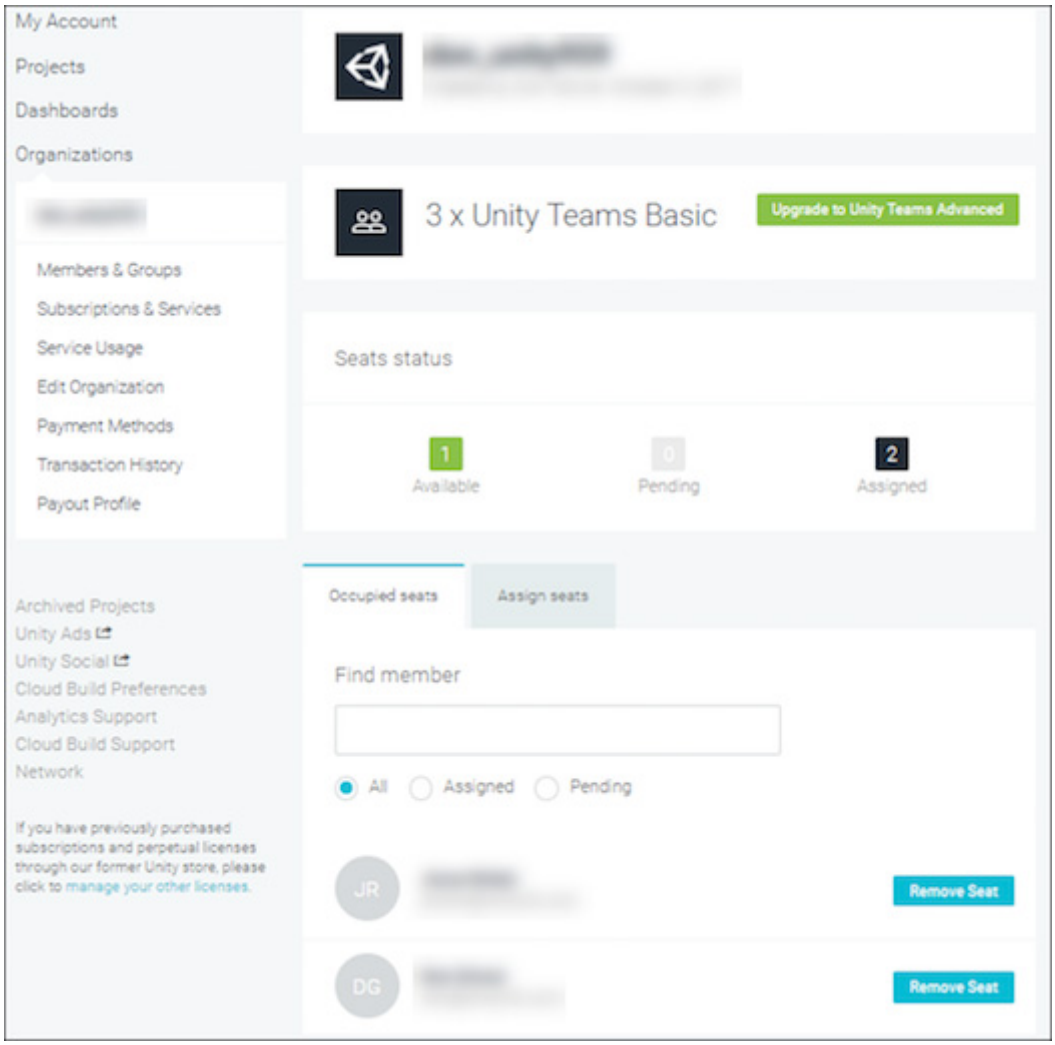
Select your organization.

Click **Subscriptions & Services.**

Select your Unity Teams subscription and click **Manage seats**.



This opens a page which allows you to assign Unity Teams seats to members of your organization. Use the **Occupied seats** tab to review team members to whom you have assigned a Teams seat. Use the **Assign seats** tab to add team members with registered Unity IDs, using the email address they used to register for a Unity ID.



2017–12–01 Page published with no editorial review

2017–12–01 - Service compatible with Unity 2017.1 onwards at this date but version compatibility may be subject to change.

New feature in 2017.1

Viewing history

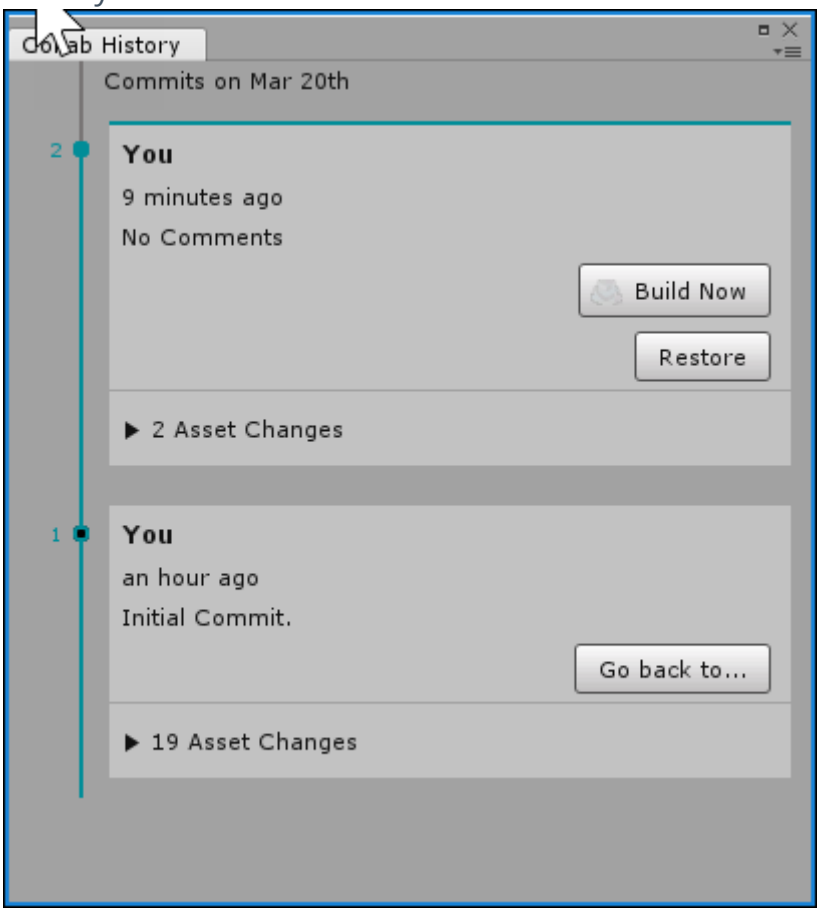
Leave feedback

You can view the entire history of the Project and any changes made.

To view the Collaboration history:

On the Unity Editor **toolbar**, click the **Collab** button.
To open the **Collab History** panel, click the **History** button (shown below), or select **Window** > **Asset Management** > **Collab History**.

History button



Collab history window

In the Collab History window, you can restore previous versions of your Project. For more information, see Restoring previous versions of a Project.

Enabling Cloud Build with Collaborate

Leave feedback

Unity Cloud Build is a continuous integration service for Unity Projects. For more information, see Unity Cloud Build.

Before you can use Cloud Build with Collaborate, you must enable Collaborate on your Unity Project. For more information, see Setting up Unity Collaborate.

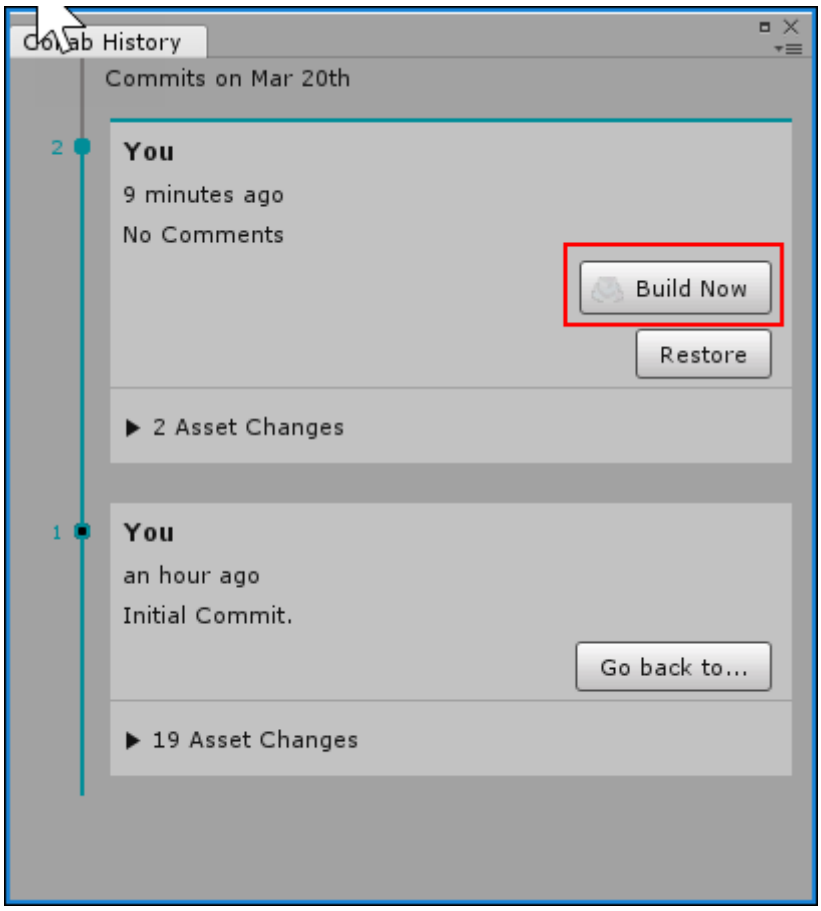
To enable Cloud Build with Unity Collaborate:

On the Unity Editor menu bar, select **Window** > **General** > **Services**.

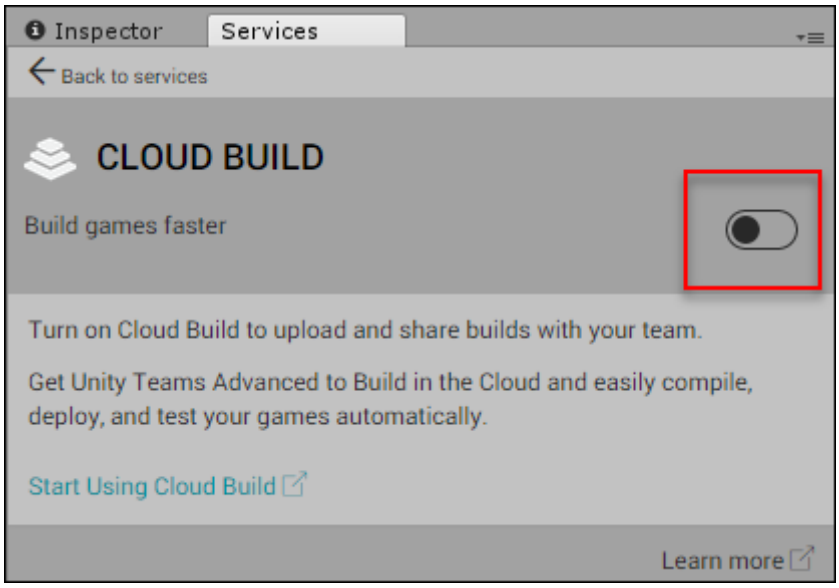
In the **Services** window, click the **Collaborate** tab.

In the **Collaborate** window, click **Open the history panel**.

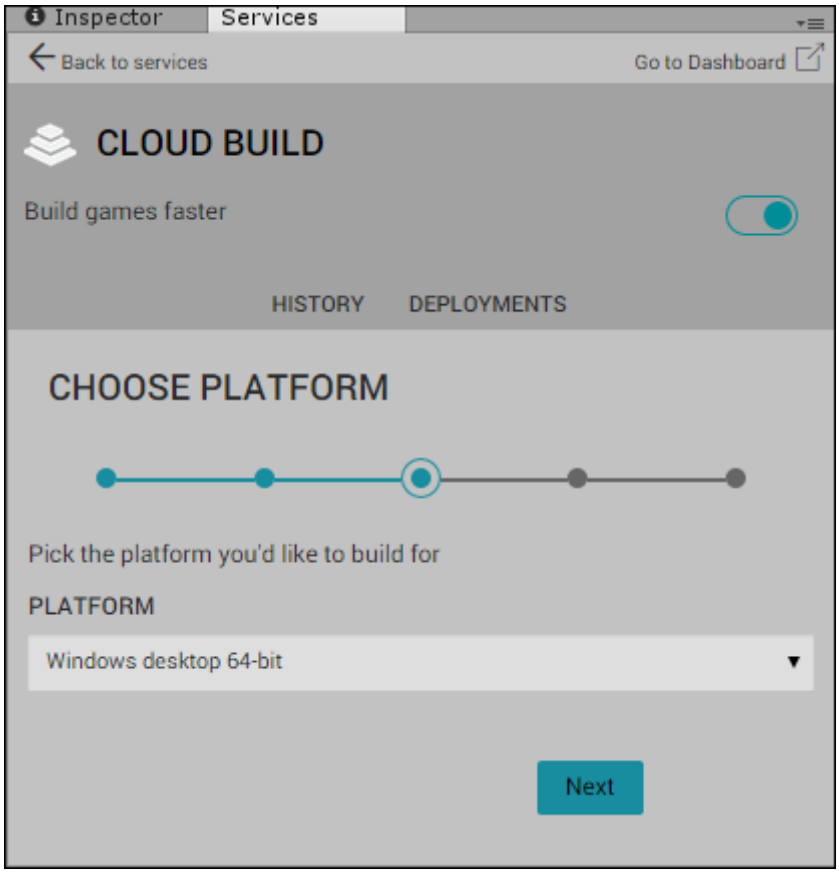
To open the **Cloud Build** window, click the **build now** button.



To enable Cloud Build, click the **Build games faster** toggle.

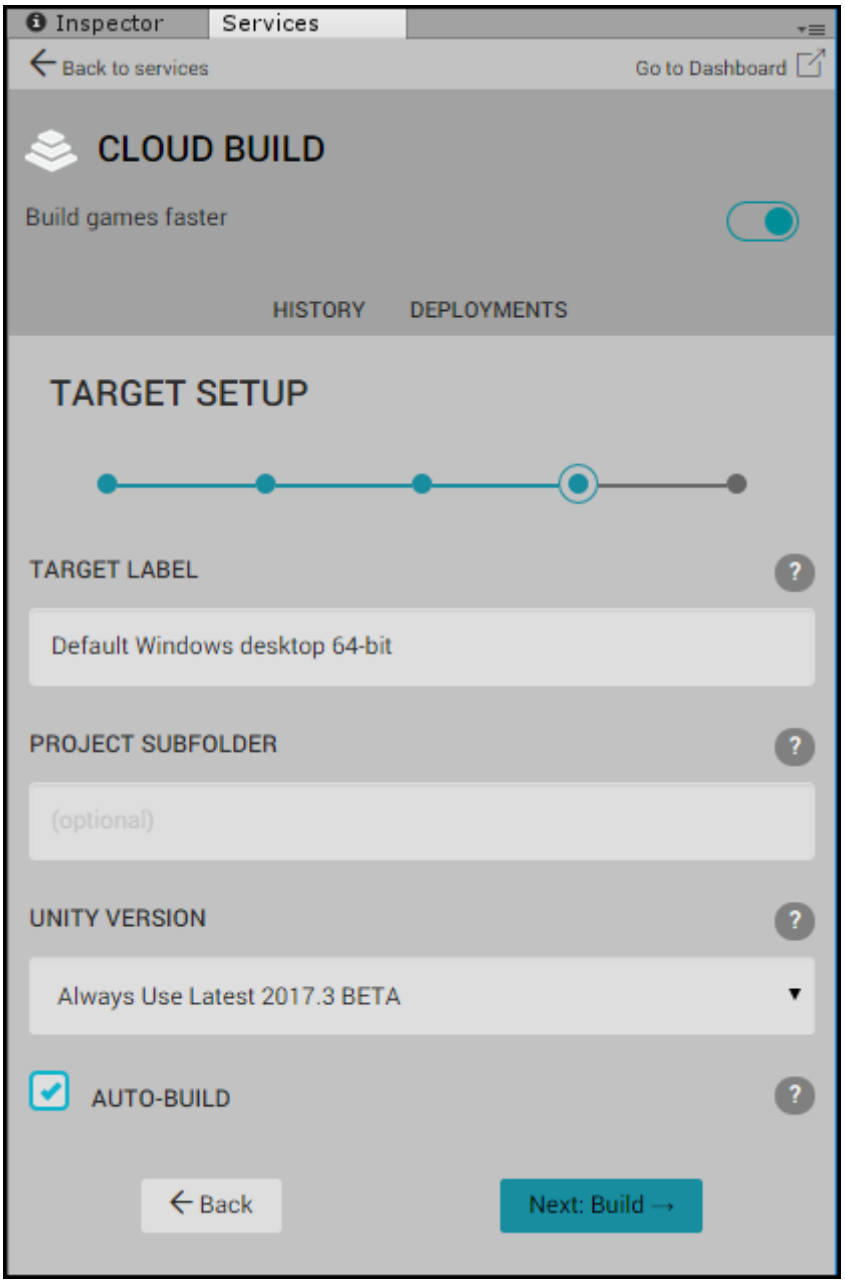


From the **PLATFORM** drop-down menu, select the build platform.



Click **Next**.

In the **TARGET SETUP** window:
In the **TARGET LABEL** textbox, enter a name to identify the build.
From the **UNITY VERSION** drop-down menu, choose the Unity version to use for the build.
If your root folder does not contain your **Assets** and **Project Settings**, specify the location in the **PROJECT SUBFOLDER** textbox.



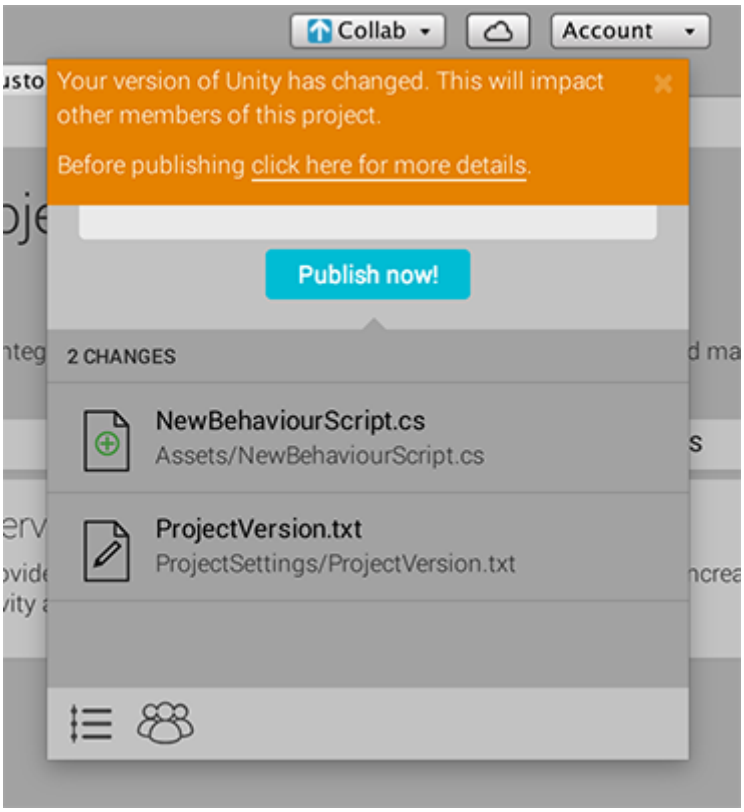
To start the initial build of your Project, click **Next: Build**.

Managing Unity Editor versions

Leave feedback

Unity Collaborate verifies that the version of Unity used for the update that you are publishing matches the version of Unity used to published the latest update shown in the Collab History window. If the version does not match, Unity displays the following warning:

```
Your version of Unity has changed. This will impact other members of this projec
```



To ensure that there are no conflicts in features and functionality, all team members should use the same version of the Unity Editor. Additionally, when a team member opens a project from a previous version of Unity in a more recent version, Unity updates the Project files to the new version. These files might not be compatible with earlier versions of Unity.

When a new version of Unity becomes available, it is important that all members of the team are able to upgrade at the same time.

For more information on the impacts of upgrading a Unity project, see the Unity Upgrade Guides.

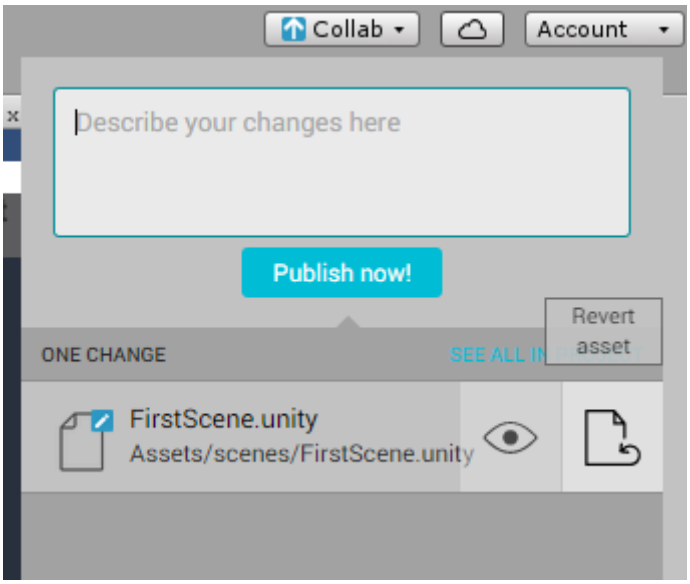
Reverting files

Leave feedback

Some changes made to a working copy of a file might not be suitable for publishing to the project.

If you have local changes you do not want to publish, click the **Revert asset** button next to the changed file. This removes any local changes you have made to that file.

Revert asset button



To confirm the removal of the local files, click **Yes** in the file revert dialog box.

See also

Restoring previous versions of a Project

Resolving file conflicts

Leave feedback

A conflict occurs when another team member makes changes to an **Asset** that you have also changed, and you try to update your Project with those changes. Collaborate automatically attempts to merge files and informs you if the attempt fails due to a conflict. For text files, Collaborate merges files using the currently configured merge tool. For **Scenes** and other files using the YAML format, Collaborate uses Unity's Smart Merge tool to perform the merge.

You can reduce the occurrence of conflicts by publishing small changes often, and communicating with your teammates. The In-Progress feature can help you reduce conflicts on scenes and **prefabs**. For more information, see In-Progress edit notifications.

Configure your merge tool

You can set your external merge tool in the preference settings of the Unity Editor:

On the Editor menu bar, select **Edit** > **Preferences**.
Click **External Tools** and, from the **Revision Control Diff/Merge** drop-down menu, select an external merge tool.
The following merge tools work well with Collaborate:

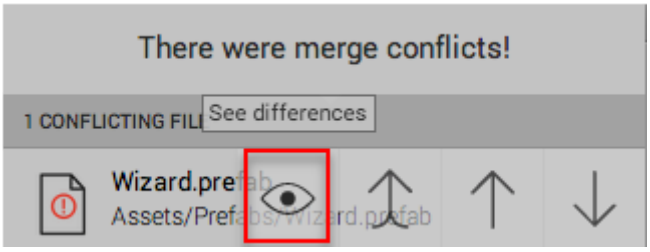
Apple File Merge
WinMerge
Beyond Compare 4
SourceGear DiffMerge
TkDiff
You might experience issues using other external merge tools.

Working with conflicts

After merging files, if there are any Assets with conflicts, Unity displays a red warning icon on the **Collab** button in the Editor **toolbar**. To view a list of Assets with conflicts, click the **Collab** button.

When you hover over an Asset in the list, the conflicts dialog shows four actions for each Asset with a conflict. You can:

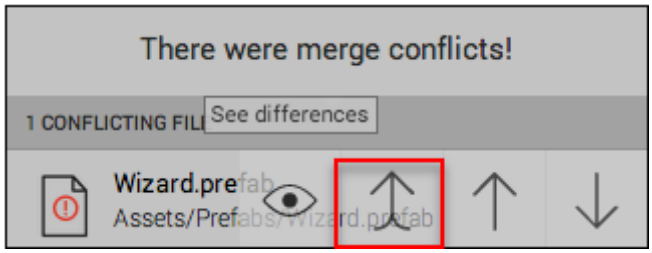
See differences, to view the conflicting changes.
Launch external tool, to open an external merge tool.
Choose mine, to publish your changes, overwriting the currently saved version of the Asset .
Choose theirs, to delete your changes and use the version of the Asset currently in the cloud.
To view a conflict, hover over the conflicted file and click the eye icon.



Eye icon

Note: You can view changes of prefabs, **scripts**, and all binary files; however you cannot view changes to image files such as .pngs.

You can use an external merge tool to resolve conflicts on prefabs, scripts, and other text-based files. To resolve the conflict using an external merge tool, click the double-up arrow.

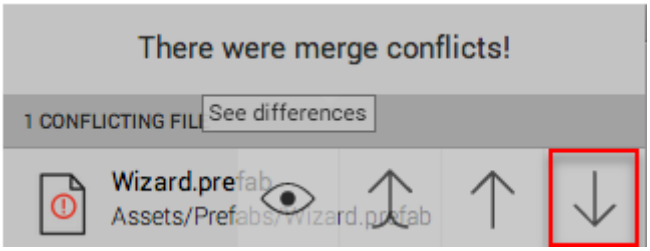


Double up arrow

To replace the version of the Asset in the cloud with your version of the Asset, click the single up arrow.

Up arrow

To discard your version of the Asset and apply the version in the cloud to your local project, click the down arrow.



Down arrow

• 2017–05–16 Page amended with no editorial review

Excluding Assets from publishing to Collaborate

Leave feedback

In some cases, you might want to exclude certain Assets in your Project from publishing to the cloud. Collaborate uses a gitignore file to exclude files from publish. To exclude Assets from publish, add them to the provided *.collabignore* file in the root of your Project folder. This lists the files and folders to exclude during a publish to Collaborate.

To add your own exclude rules to the *.collabignore* file:

Read the GitIgnore documentation on Git-SCM.com.
Edit the *.collabignore* file to add your new rules.
Start the Unity Editor (or restart the Editor if it is already running).
Publish your *.collabignore* file changes in Collaborate to share your exclusions with the rest of your team.

Note: For local edits to the *.collabignore* file to take effect, you must restart the Unity Editor.

Note: If you exclude a file that is already tracked in Collaborate, its existing file history is preserved.

There are Project files and folders that you can never exclude from Collaborate using the *.collabignore* file. These are:

The *.collabignore* file.
The Assets folder (although you can exclude specific files or folders within the *Assets* folder.)
The *Project Settings* folder.
Any *.asset* file inside the *Project Settings* folder.
ProjectVersion.txt within the *Project Settings* folder.

See also

Setting up Unity Collaborate

Publishing individual files to Collaborate

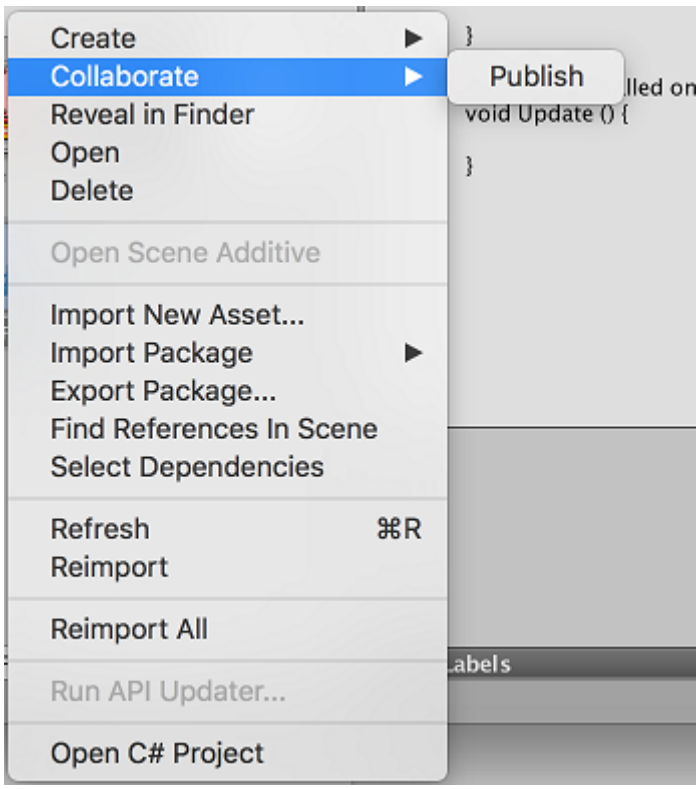
Leave feedback

Colloborate allows you to partially *publish* a Project. A partial publish allows you to publish only those files that you are ready to share with the rest of the team or just save work to the cloud for safekeeping. So, instead of publishing all the changed Assets in a Project, you can publish specific Project to Collaborate.

To publish specific Assets:

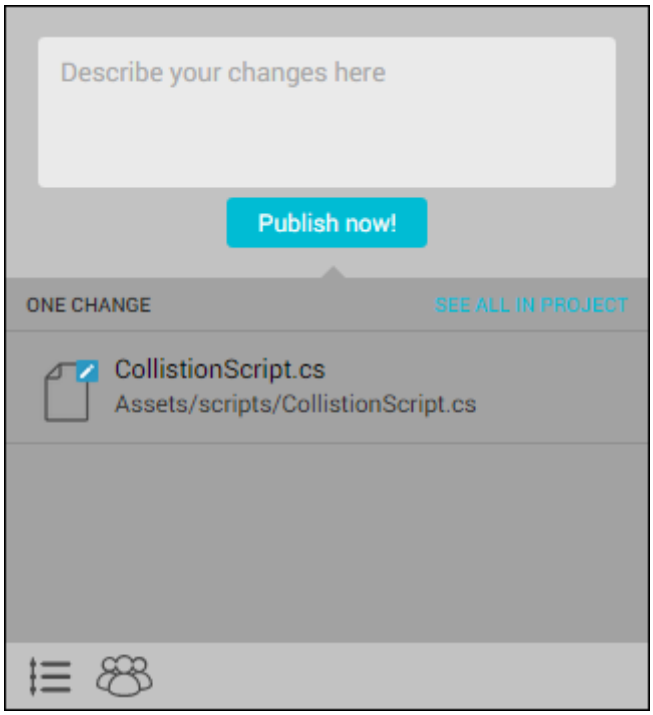
In the Unity Editor, with your Project open, right-click on the new or modified Asset, sets of Assets, or folder to publish.

Select **Collaborate** > **Publish**.



Publish Asset menu

In the Publish window, enter a publish comment and then click **Publish now**.



Publish Asset menu

Restoring previous versions of a project

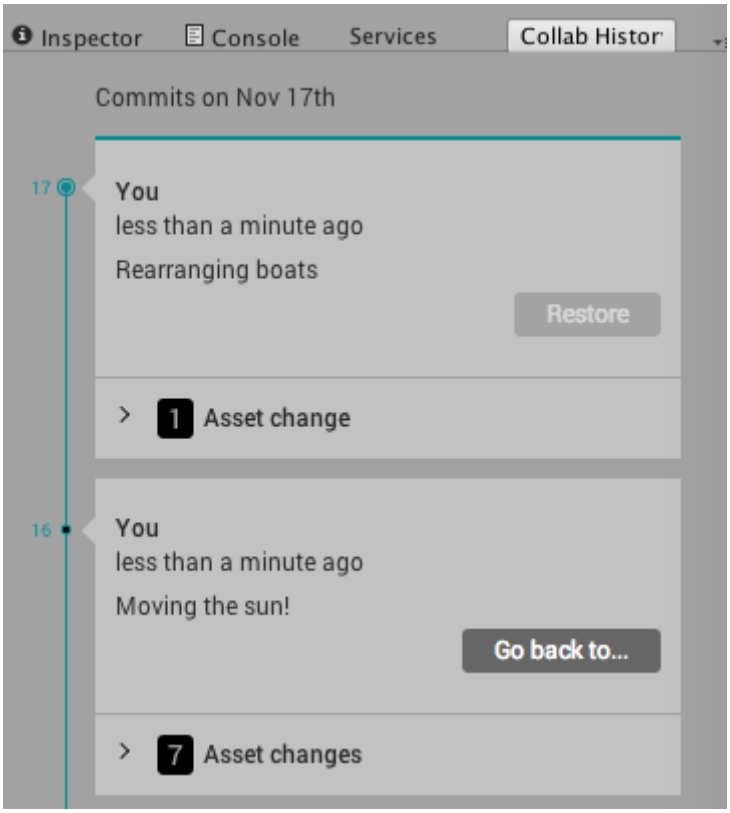
Leave feedback

There are times when you need to revert your entire Project to a previous version. When this occurs, you can use the **Restore** and **Go Back** functionality to retrieve a previous version of your Project. You can then publish the retrieved Project as the current version in the cloud. Collaborate ensures that all versions of your Project are intact, even after a restore, and that you can access them at any time in the development cycle.

To open the **Collab History** window, on the Unity Editor menu bar, select **Window** > **Asset Management** > **Collab History**.

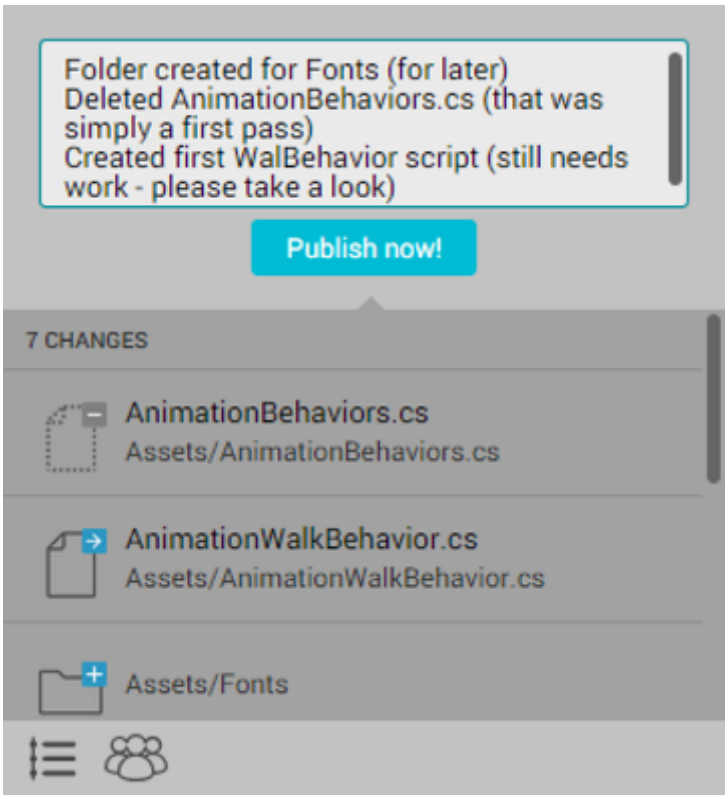
To remove all local changes and restore the latest version from the cloud, on the head node of the change history, click the **Restore** button.

To restore a version published earlier than the current version in the cloud, click the **Go back to** button.



Restoring replaces all local Project files with the files from the older version. The files are now listed as modified in the Collaborate toolbar. A blue **plus sign (+)** on a file or folder indicates that it is modified. A blue **arrow (->)** on a file or folder means it is new (see image below).

To publish the restored version as the current revision in the cloud, click the **Publish now!** button.



See also

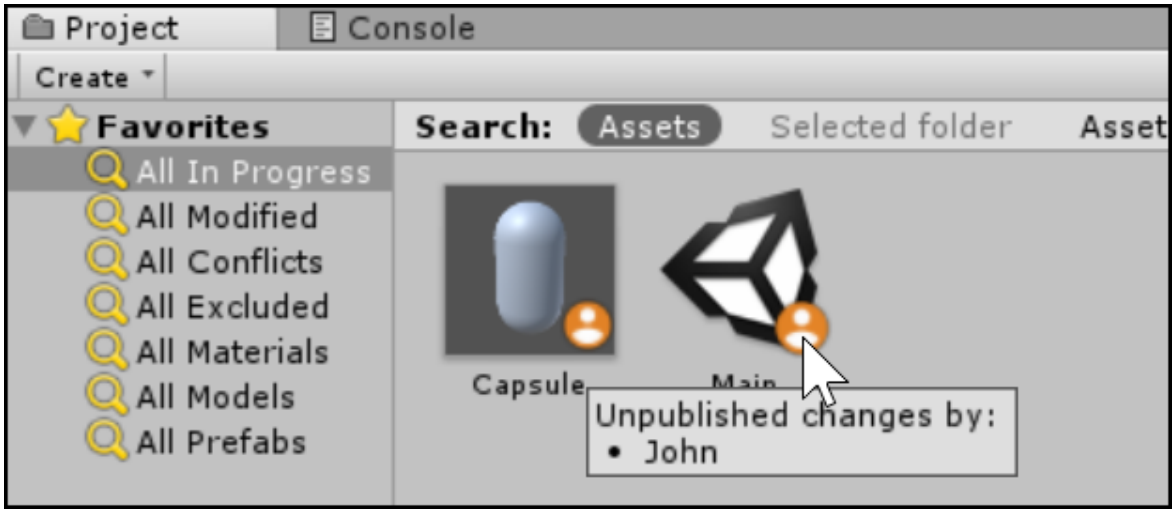
Reverting files

In-Progress edit notifications

Leave feedback

Unity Collaborate constantly monitors the changes team members make to your Project. When a team member saves changes to a **Scene** or **Prefab** on their local machine, Collaborate notifies you that the **Asset** has been changed by displaying an In-Progress badge on the **Asset**.

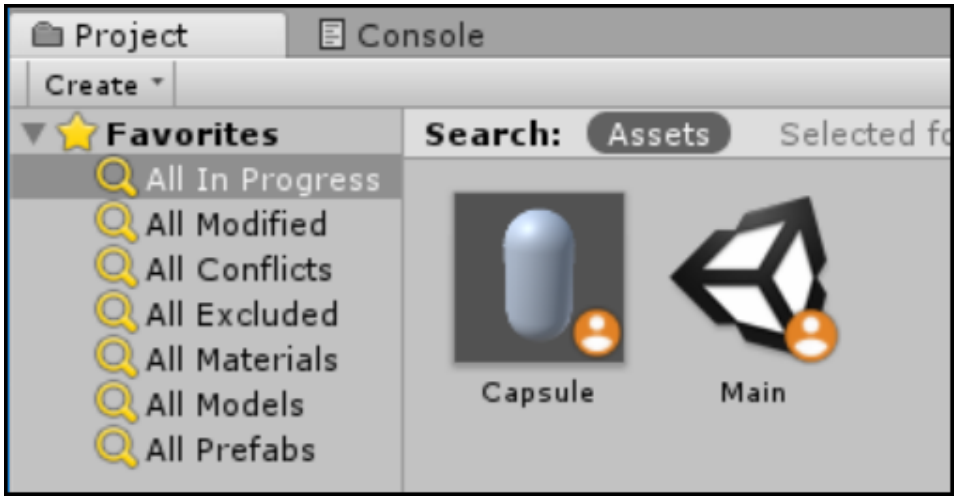
To view who is working on the Scene or Prefab, hover over the badge.



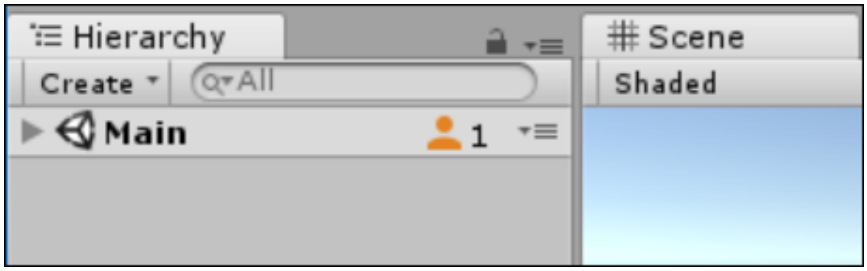
In-Progress for Scenes

For Scenes, the In-Progress badge is visible in two locations in the Unity Editor:

On the file icon in the **Project** browser.



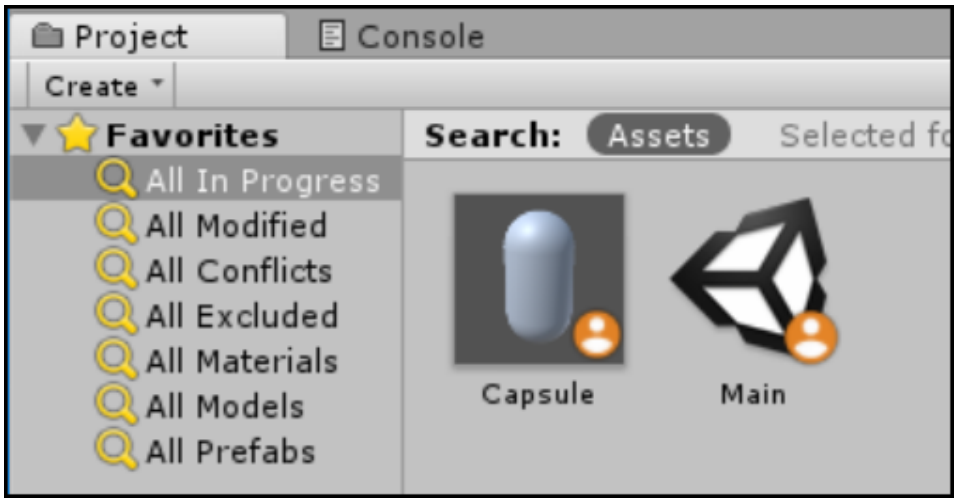
In the **Hierarchy** window.



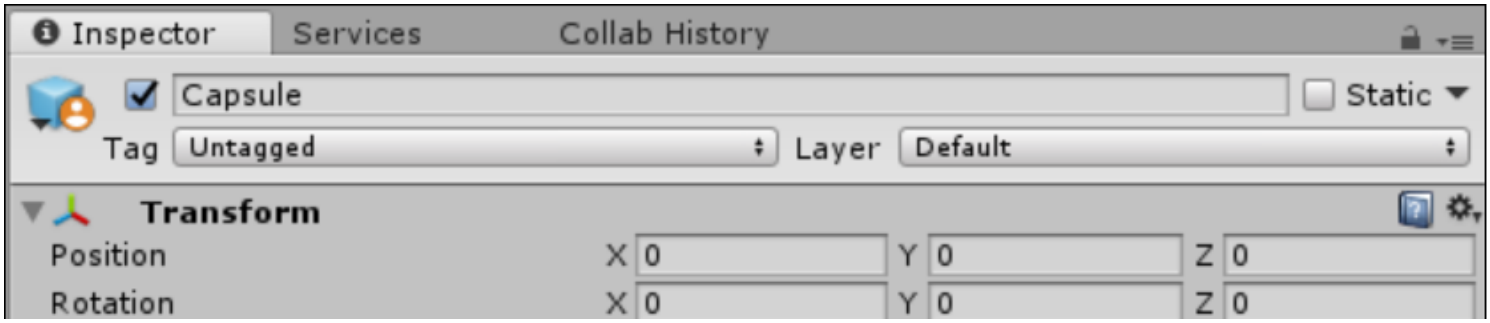
In-Progress for Prefabs

For Prefabs, the In-Progress badge is also visible in two locations in the Unity Editor:

On the file icon in the Project browser.



In the **Inspector** window.



When a team member publishes their changes, Unity removes the icon and counter.

New feature in Unity 2017.1

Managing cloud storage

Leave feedback

The Collaborate service uses cloud storage to backup your entire Project in the cloud and make it accessible anywhere.

This topic covers:

Determining how much storage your Projects are using
Adding more storage
Cloud storage data retention
Archiving Projects
Archiving Projects for Collaborate

Determining how much storage your Projects are using

Cloud storage is calculated on a per Organization basis. Unity calculates the amount of cloud storage that your Organization is using by summing the size of the latest versions of all your Organization's Collaborate Projects. Only the latest version of your project counts towards your storage limit. Historical (non-head) versions of your Projects do not count towards cloud storage.

To view your current storage usage:

Sign into the Unity Developer Dashboard.
Select a Project in the Organization whose storage you want to view.
In the left navigation window, click **Settings**, then click **Usage**.
In the main Project window there are tiles that show:

Cloud storage - How much storage the current Project is using compared to the rest of the Projects in your Organization, and your entire cloud storage space.
Percent of storage per project - How much storage is used by each Project in your Organization.
Storage used by <Project name> over time - How much storage the current Project has used since it was created, until the current time.

Adding more storage

You can obtain more cloud storage in two ways:

Upgrade to Unity Teams Advanced, which also provides 25GB of cloud storage.
Purchase Storage Packs on the Unity ID dashboard. Each storage Pack adds 25GB of cloud storage to an Organization.

Upgrading to Unity Teams Advanced

To upgrade to Unity Teams Advanced, visit the Unity Teams page at unity3d.com/teams. On the Unity Teams page you can learn more about the features of Unity Teams Advanced and then upgrade.

Purchase storage packs

To add storage packs:

Sign in to the Unity ID dashboard.
In the left navigation bar, click **Organizations**.
In the main **Organizations** window, select an Organization.
In the left navigation bar, click **Service Usage**.
In the **Usage overview** window, click the **Manage storage** button.
On the **Add cloud storage** dialog, select the amount of storage that you need, and then click the **Continue to purchase** button. If you have only one Organization, proceed to step 6.

In the **Checkout** window, enter your payment information and then click **Pay now**.

Modifying your contracted amount of storage

Sign in to the Unity ID dashboard.
In the left navigation window, click **Organizations**.
In the main **Organizations** window, select an Organization.
In the left navigation bar, click **Service Usage**. The Service Usage window displays your contract period and the number of storage packs that you are contracted for.
In the **Usage overview** window, click the **Manage storage** button.
On the **Add cloud storage** tab, enter the number of storage packs to add.
Click the **Next** button.
In the **Review order** window, review the order information and then click the **Confirm purchase** button.

You can reduce the number storage packs on the **Manage subscription** tab:

To remove all contracted storage packs, turn off **Automatic Renewal**. When automatic renewal is turned off, all storage packs are removed on the renewal date.
To reduce the number of storage packs that are renewed, enter the number of storage packs to renew in the **Select the number of storage packs you want to renew upon renewal** field.

Then click the **Save** button.

Cloud storage data retention

If you're using Unity Teams Advanced, Unity does not delete any version history for your Projects.

If you're using Unity Teams Basic, Unity stores your data in cloud storage for at least 1 year. Unity reserves the right to delete Project history older than 1 year. You always have access to the latest version of your Project, but you only have access to the last 90 days of your version history. If you're using Unity Teams Basic and want full access to the last 1 year of your history, you must upgrade to Unity Teams Advanced. When you upgrade to Unity Teams Advanced, you have access to your full history going forward.

Reducing your cloud storage usage

You have three options to reduce your cloud storage usage:

Delete or reduce the size of **Assets** in your Project. It is important to remember that Project history does not count toward your cloud storage usage.
Archive unused Projects. See Archiving Projects.
Archive Projects for Collaborate only. Archiving a Project for Collaborate only allows you to free cloud storage space, while still being able to use other Unity services such as Ads, **Analytics**, and Performance Reporting. See Archiving Projects for Collaborate.

Archiving Projects

Archiving a Project automatically archives any Collaborate cloud storage associated with the Project and Unity services are no longer accessible.

To archive a Project:

Sign into the Unity Developer Dashboard.
Select the Project to archive.
In the left navigation bar, click **Settings**, then click **General**.
At the bottom of the main window, read the important note and, if you are certain you want to archive the Project, click the **Archive my Project for ALL Unity services** button.

In the **Are you sure you want to archive this Project** dialog, enter the name of the Project and then click the **Yes, archive it** button.

To unarchive a Project:

Sign into the Unity Developer Dashboard.
At the top right of the main window, under the **CREATE NEW PROJECT** button, click the **View Archived Projects** link.
Locate the Project to reactivate, click the vertical ellipsis to the right of the Project name and then select **Unarchive Project**.

Archiving Projects for Collaborate

To archive a Project for Collaborate only:

Sign into the Unity Developer Dashboard.
Select the Project to archive.
In the left navigation window, click **Collaborate**, then click **Storage**.
In the main window, click the **ARCHIVE** button.

To unarchive a Project for Collaborate:

Sign into the Unity Developer Dashboard.
Select the Project to unarchive.
In the left navigation window, click **Collaborate**, then click **Storage**.
In the main window, click the **UNARCHIVE** button.

2018–07–16Page published with editorial review

Moving your Project to another version control system

Leave feedback

To export your Project as a zip file that you can import into another **version control** system:

Sign into the Unity Developer Dashboard.
Select the Project to export.
In the left navigation window, click **Collaborate**, then click **Storage**.
In the main window, click the **START EXPORT** button.
When export is complete, click the **DOWNLOAD EXPORT** button. A zip file downloads onto your local machine.
The zip file contains all versions of all files stored in the Project's repository, ready for import into another version control system.

Note: Exporting a Project does not reduce your storage usage.

2018–07–16 Page published with editorial review

Collaborate troubleshooting tips

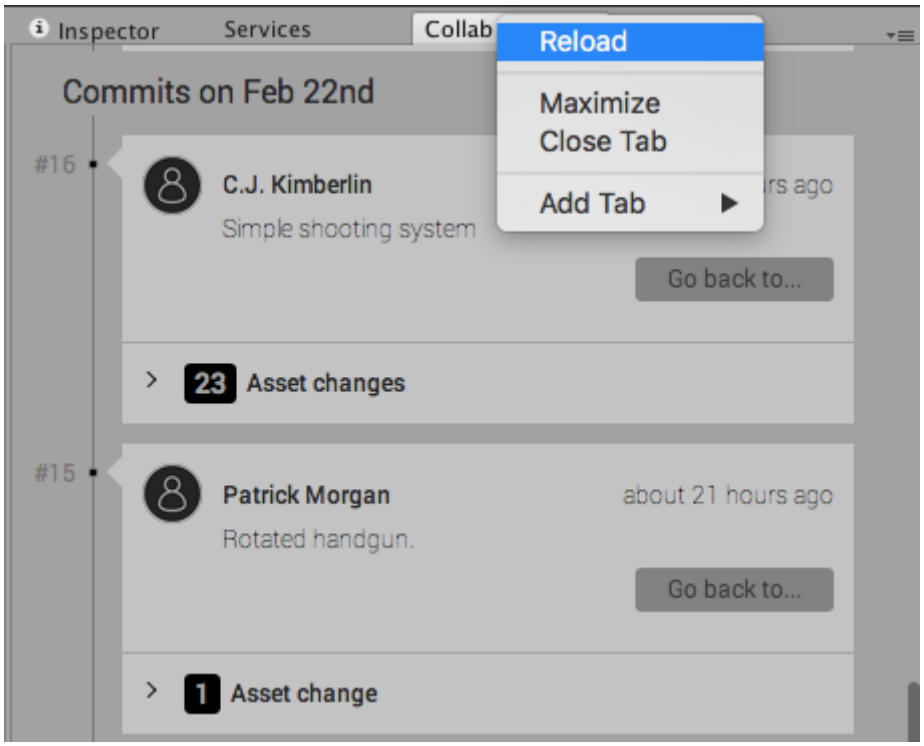
Leave feedback

This page details some simple troubleshooting tips for the most common problems that can arise while using Unity Collaborate. For further assistance, visit the Collaborate section of the Unity forum.

Out-of-date Collab History information

Issue: The information in the **Collab History** window appears to be out of date (for example, if the time next to the latest publish doesn't match when the most recent changes were pushed).

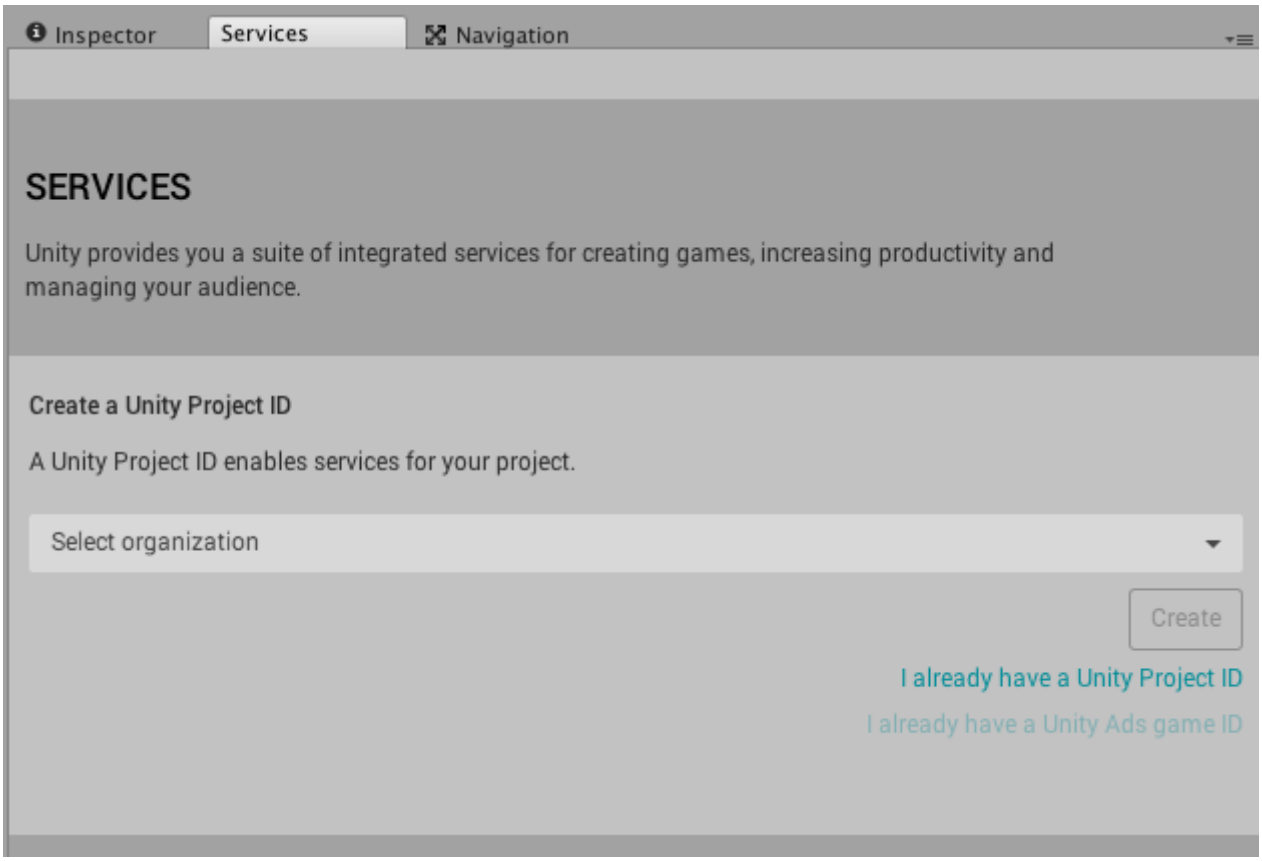
Resolution: Right-click the **Collab History** tab and select **Reload** to retrieve the most recently updated information.



Project state problems

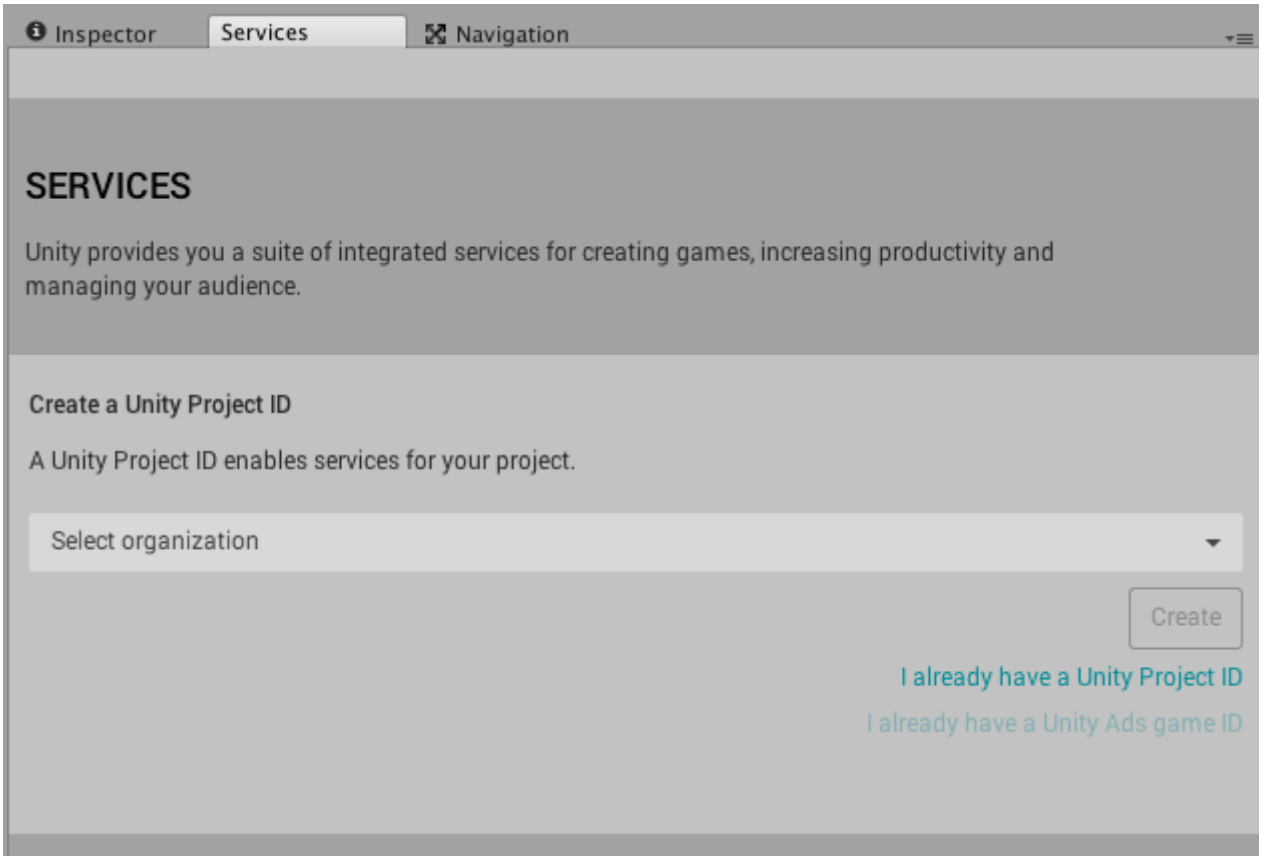
Issue: Your local Project state is the state you want, but you are experiencing conflicts or other problems with the Cloud Project state.

Resolution: Re-link the Project. Open the **Services** tab and navigate to **Settings**. From here, click **Unlink project**, then re-link the Project with a new ID to submit a clean new version of your Project. To do this, click **Select organization** in the Services panel and follow the steps to assign or create an ID.



Issue: You are experiencing conflicts or other problems with the local Project state, and can revert your local version to a version of the Project in the Cloud.

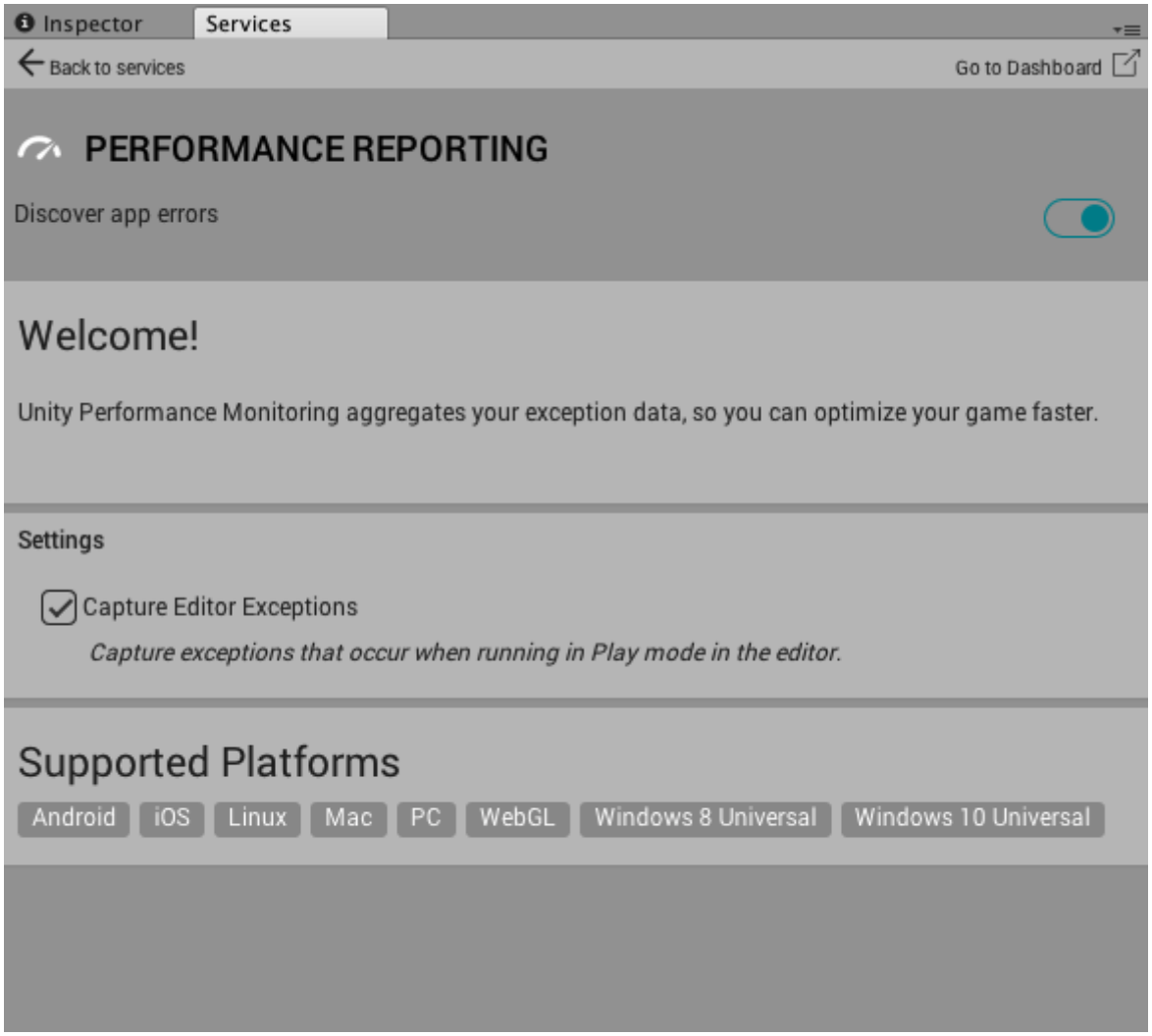
Resolution: Revert to the Cloud version using the instructions found in Restoring previous versions of a Project.



Unity Performance Reporting

Leave feedback

Unity Performance Reporting captures and aggregates exception data and native crash reports, to help you understand what's happening during run time and to optimize your project faster. Exceptions are updated to the Services Dashboard while the game is running as a completed build, and while in Play Mode within the Editor.



Services Dashboard

For information on enabling Performance Reporting to track crashes, including crashes in native user code, engine code, and plug-ins, see Setting up Performance Reporting.

Native crash reporting availability

Native crash reporting is available to Unity Plus and Pro users as part of the Unity Performance Reporting service.

Platform	Available starting with version
iOS	5.6
Android	2017.2
MacOS	2017.3
Windows	2018.1

iOS crash reporting

When you use IL2CPP and have the iOS Player Settings **Script Call Optimization** setting set to **Fast but no exceptions**, iOS crash reporting captures crashes due to uncaught managed exceptions.

For more information on the **Script Call Optimization** settings, see iPhone Optimization.

Reports

Reports show the native stack trace of the crashed thread. If the crash is caused by an uncaught managed exception, the reports also show the managed stack trace of where the exception originates from, along with the native stack trace.

Note that iOS crash reporting cannot capture crashes caused by the application running out of memory and then being closed by the operating system.

View your crashes in the Unity Cloud Developer Dashboard. Here, select your game and select **Enable Game Performance** (see Image A).

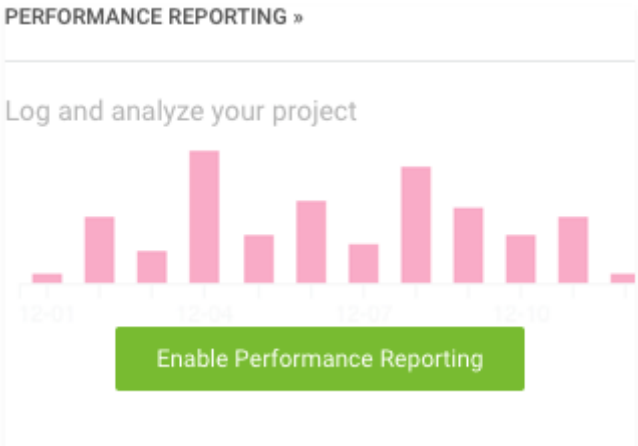


Image A: The Game Performance option on the Unity Cloud Developer Dashboard

Example

The example report below demonstrates what a report for a crash due to a managed exception looks like:

```
Managed Stack Trace:

at CrashTests.CrashNextUpdate () [0x00000] in <filename unknown>:0

at CrashTests.Update () [0x00000] in <filename unknown>:0

Native StackTrace:

Thread 0 (crashed)

0 crashreporttest CrashedCheckBelowForHintsWhy() (CrashReporter.mm:106)

1 crashreporttest UnhandledExceptionEventHandler_Invoke_m689053609 (mscorlib_Sys
```

```
2 crashreporttest RuntimeInvoker_Void_t2779279689_Il2CppObject_Il2CppObject(Meth

3 crashreporttest il2cpp::vm::Runtime::CallUnhandledExceptionDelegate(Il2CppDoma

4 crashreporttest il2cpp::vm::Runtime::UnhandledException(Il2CppObject*) (Runtim

5 crashreporttest ScriptingInvocationNoArgs::Invoke(ScriptingException**) (Scrip

6 crashreporttest ScriptingInvocationNoArgs::Invoke() (ScriptingInvocationNoArgs

7 crashreporttest MonoBehaviour::CallUpdateMethod(int) (MonoBehaviour.cpp:541)

8 crashreporttest void BaseBehaviourManager::CommonUpdate<BehaviourManager>() (B

9 crashreporttest PlayerLoop(bool, bool, IHookEvent*) (Player.cpp:1721)

10 crashreporttest UnityPlayerLoopImpl(bool) (LibEntryPoint.mm:240)

11 crashreporttest UnityRepaint (UnityAppController+Rendering.mm:238)

12 crashreporttest -[UnityAppController(Rendering) repaintDisplayLink] (UnityApp

13 QuartzCore CA::Display::DisplayLinkItem::dispatch()

14 QuartzCore CA::Display::DisplayLink::dispatch_items(unsigned long long, unsig

15 IOKit IODispatchCalloutFromCFMessage

16 CoreFoundation __CFMachPortPerform

17 CoreFoundation __CFRUNLOOP_IS_CALLING_OUT_TO_A_SOURCE1_PERFORM_FUNCTION__

18 CoreFoundation __CFRunLoopDoSource1

19 CoreFoundation __CFRunLoopRun

20 CoreFoundation CFRunLoopRunSpecific

21 GraphicsServices GSEventRunModal

22 UIKit UIApplicationMain

23 crashreporttest main (main.mm:32)
```

2018–03–27 Page amended with limited editorial review

Capture Editor Exceptions setting added and **Optimize game performance** toggle label renamed to **Discover app errors** in Unity 5.5

Setting up Performance Reporting

Leave feedback

Once you have set up your Project for Unity Services, you can enable the Performance Reporting service.

How to turn on Performance Reporting

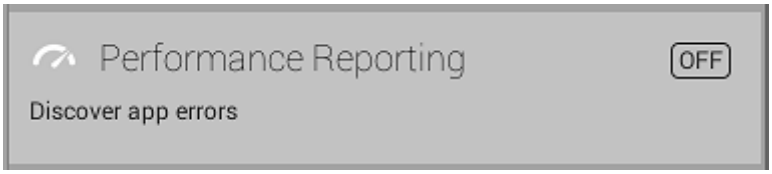


Image A: Performance Reporting listed in the Services window

To turn on Performance Reporting:

In the Unity Editor menu bar, select **Window** > **General** > **Services** to open the Services window.

In the Services window, select **Performance Reporting**.

Click the black toggle (shown in Image B) to turn on the service. It turns blue when it is on.

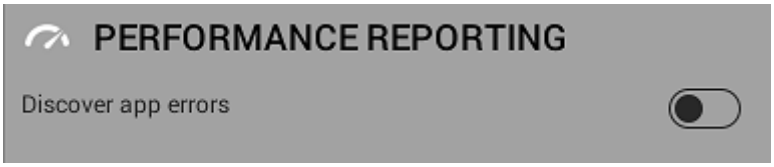
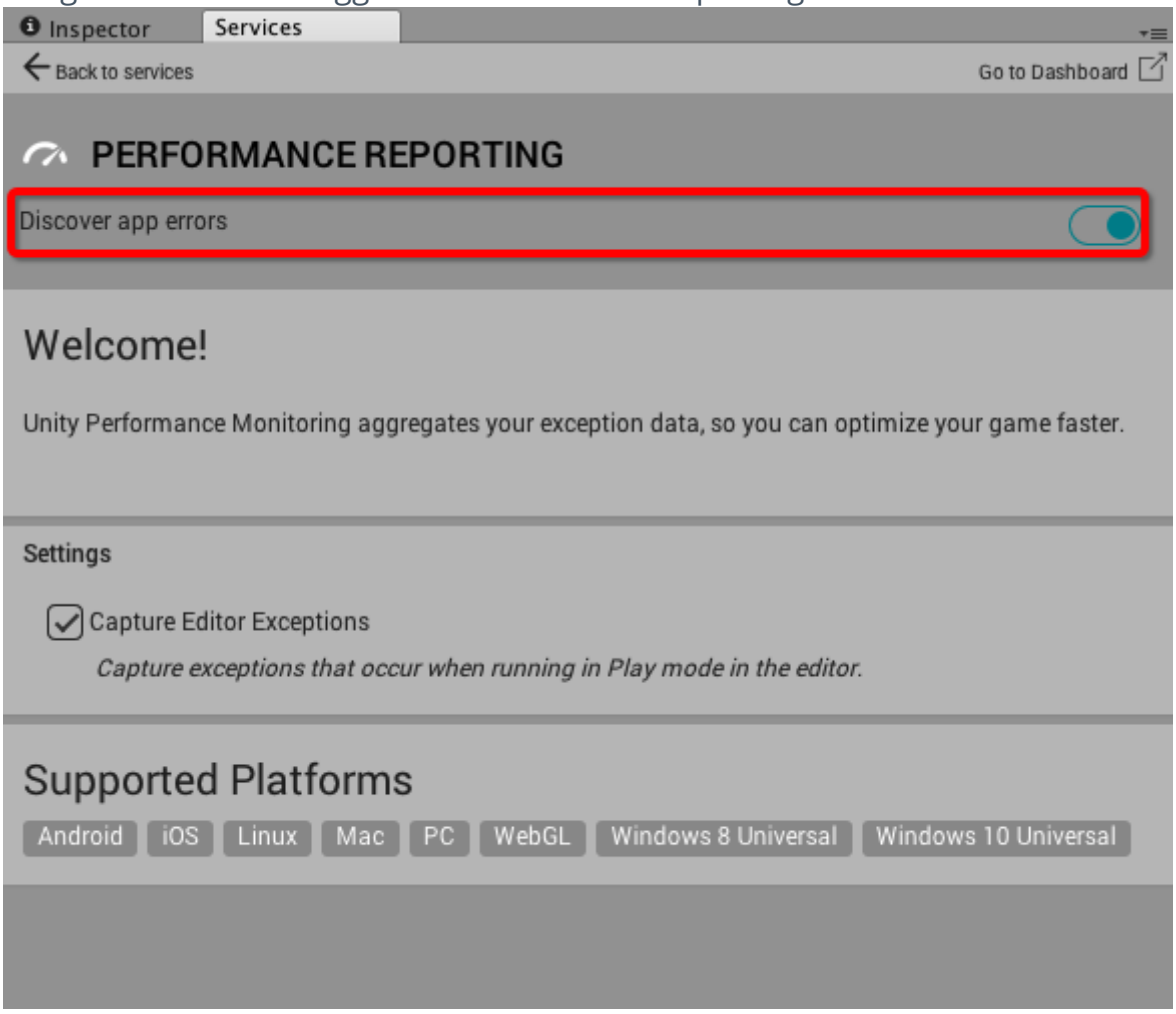


Image B: The on-off toggle for Performance Reporting



Toggle **Discover app errors** to turn on Performance Reporting

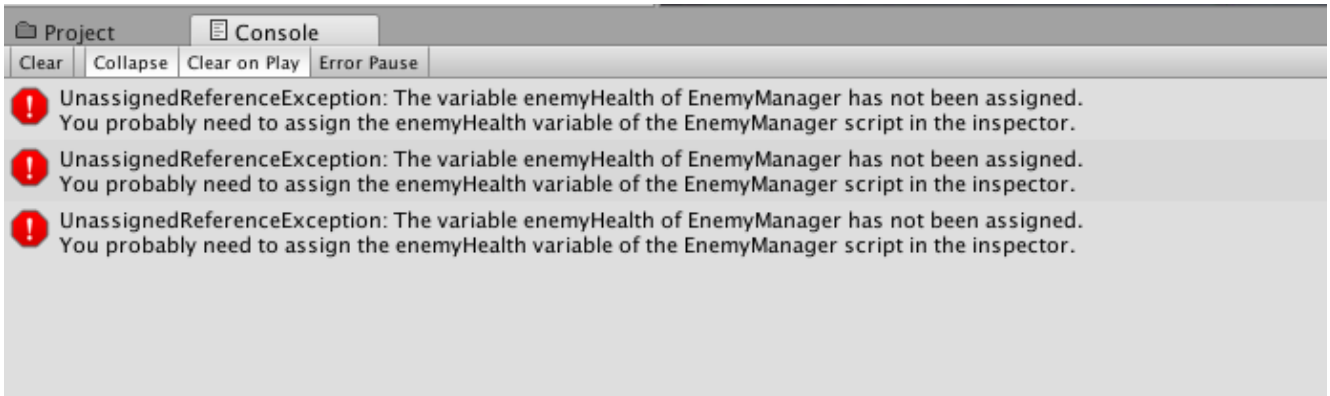
2017–09–04 Page amended with limited editorial review

Capture Editor Exceptions setting added and **Optimize game performance** toggle label renamed to **Discover app errors** in Unity 5.5

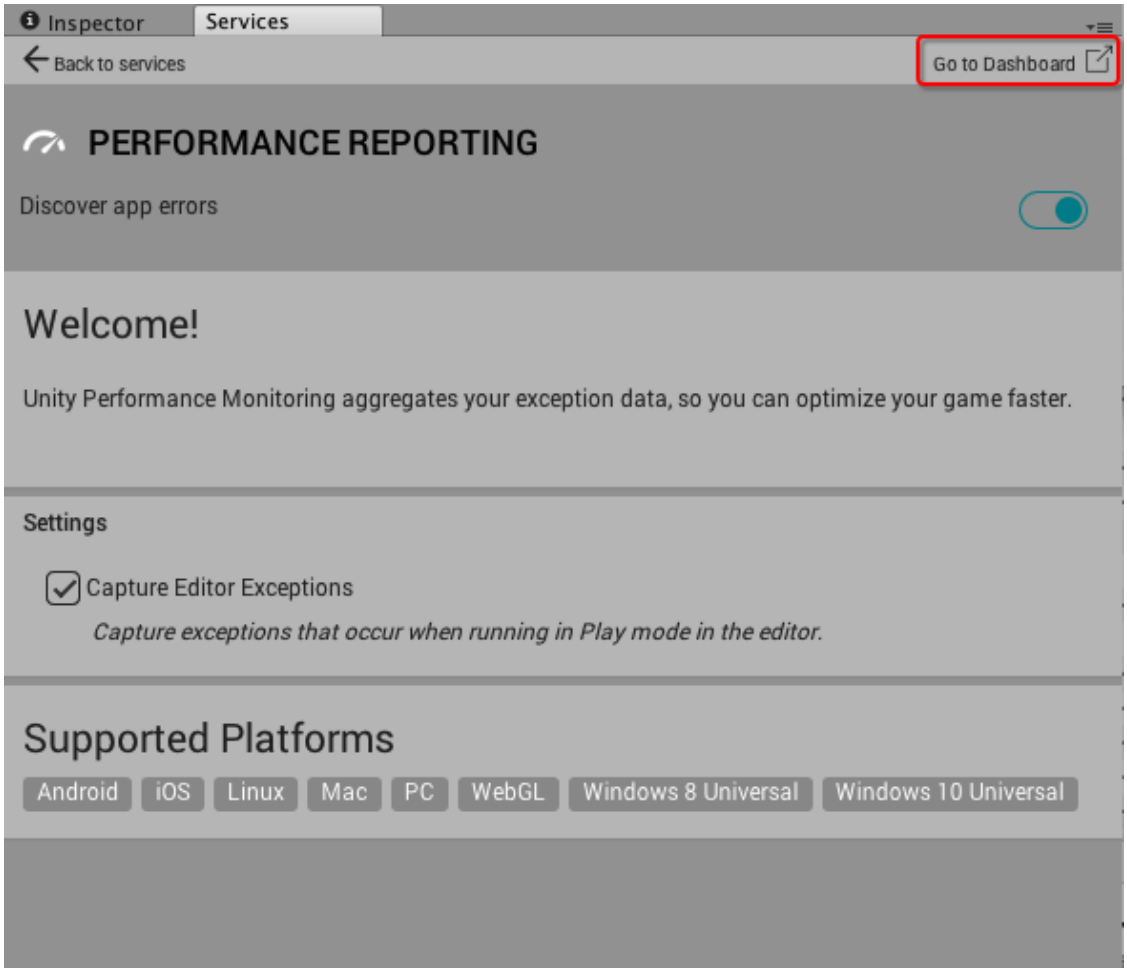
Understanding exception reports

Leave feedback

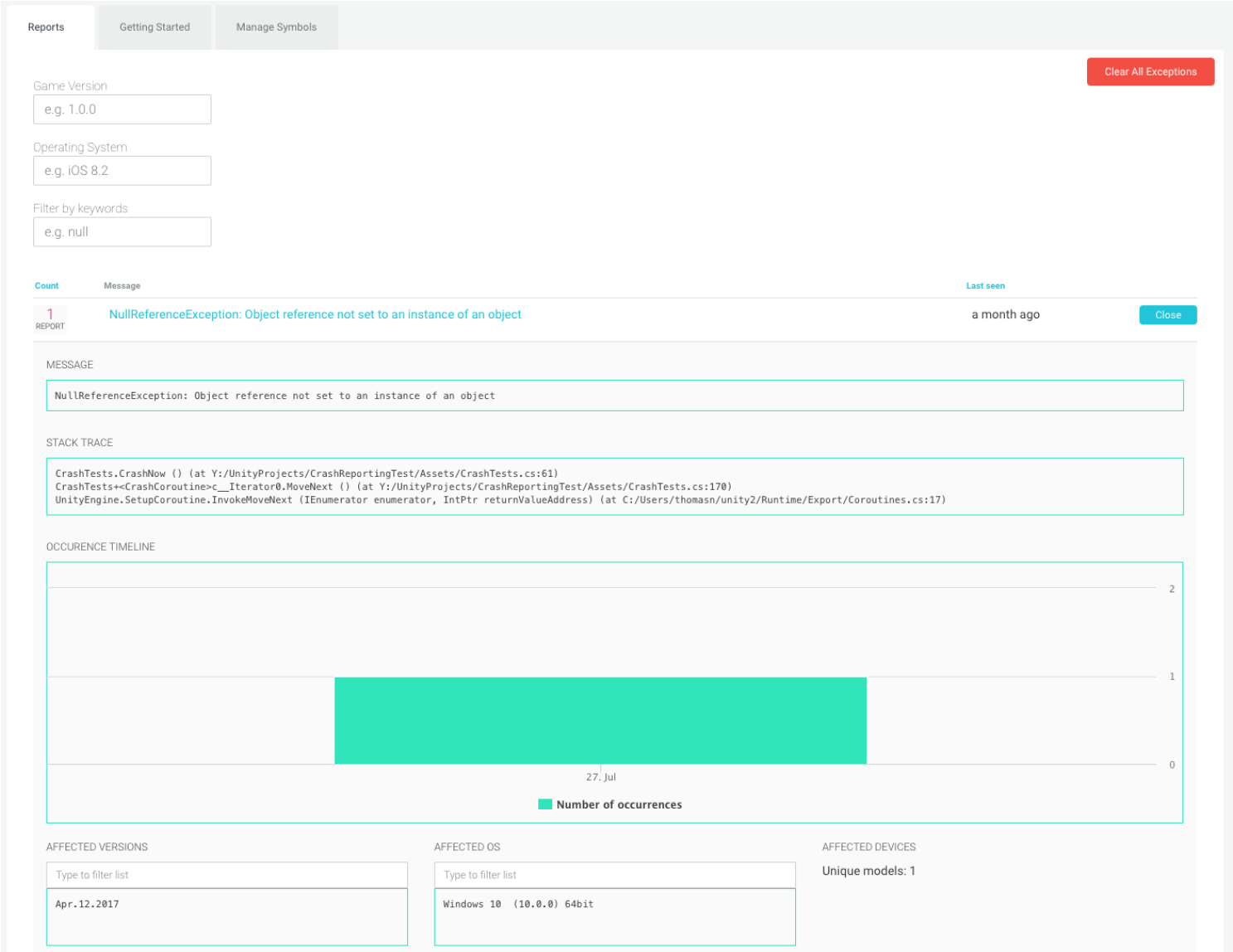
Exceptions that occur during run time are flagged up in the console as normal. With Performance Reporting enabled, these exceptions are also reported to the Performance Reporting Dashboard.



The Dashboard is accessed online via your Internet browser. To view it, open the **Performance Reporting** panel in the Unity Editor and select **Go to Dashboard**:



The Dashboard displays a list of your most recent exceptions. For more information about a specific exception, click on the exception to expand the view:



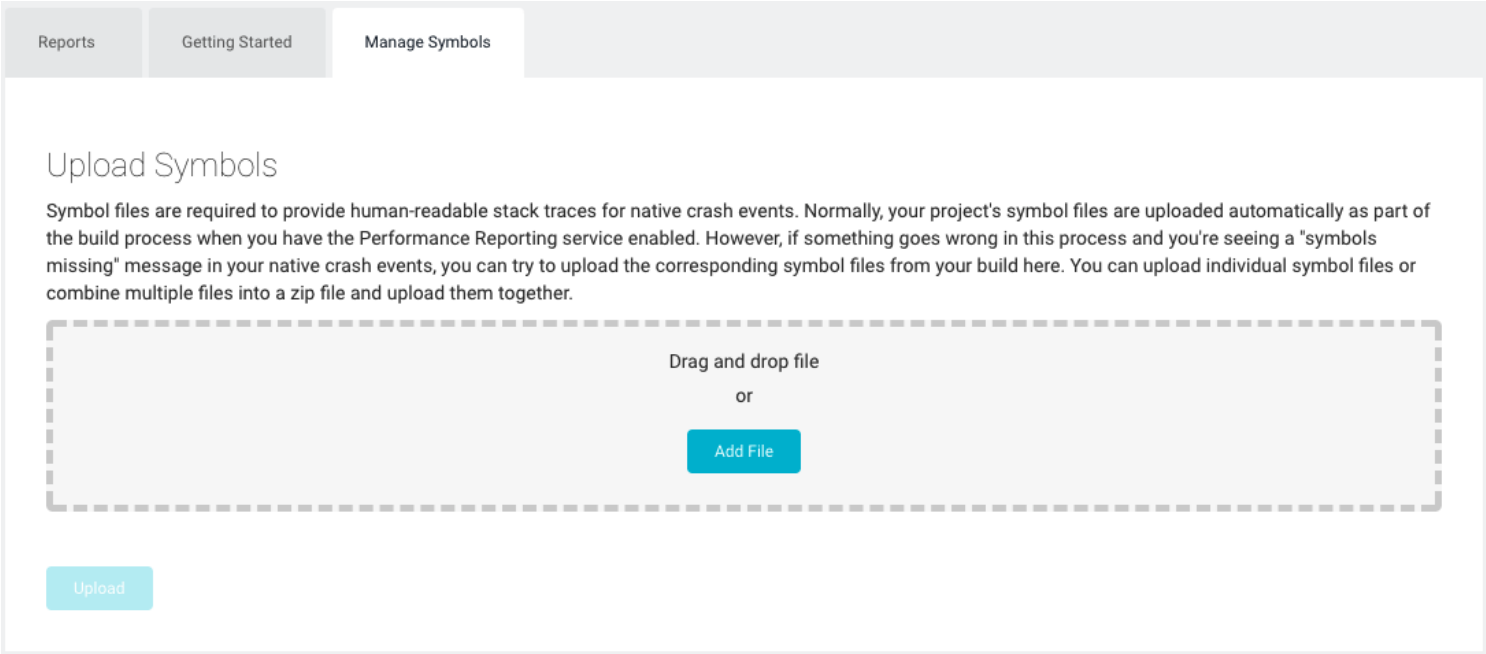
The expanded version shows you the console message, the stack trace, an occurrence **timeline**, affected versions of the game, affected operating systems, and affected devices.

Manage Symbols tab

Symbol files are required to provide human-readable stack traces for native crash events. Normally, your project's symbol files are uploaded automatically as part of the build process when you have the Performance Reporting service enabled. However, if something goes wrong in this process and you're seeing a "symbols missing" message in your native crash events, you can try to upload the corresponding symbol files from your build here. You can upload individual symbol files or combine multiple files into a zip file and upload them together.

The symbol upload processor can handle the following formats:

Mach-O: **iOS**/OSX, a .dSYM folder
ELF: Android/Linux, a .so file
PDB: Windows, a .pdb file
The UUID or GUID identifier of your symbol file must match what the dashboard is showing as missing in order for symbolication to be successful. Tools such as dwarfdump -u (OSX) and readelf -n (Linux) can help you check these. Some tools report the UUID with dashes in it, but the dashes and the casing are not important as long as the bytes of the identifier match those shown on the dashboard.



2017–09–04 Page amended with limited editorial review

Added Manage Symbols tab in Unity 5.6

Performance Reporting missing symbols

Leave feedback

Symbols map program addresses to function names. They allow the Performance Reporting service to provide a native crash stack trace with human-readable function names, instead of just memory addresses. They can be contained within the executable itself, but are usually stored in a separate file to reduce executable size and make it more difficult to tamper with the executable.

Symbol files come in different formats depending on the operating system. Apple platforms use dSYM folders, Android symbols are stored in .so files, and Windows symbols are stored in .pdb files. The Performance Reporting service can identify and process all of these formats.

The Performance Reporting service works with two separate sets of symbols:

System symbols - System symbols are generated by the operating system supplier. Unity supports symbols generated by Apple, Google and Microsoft.
Application symbols - Application symbols are generated when you build your Unity project.

Why are my symbols missing?

Symbol files have a universally unique identifier (UUID) or globally unique identifier (GUID) that must exactly match the ID of the executable. The Performance Reporting service generates the following errors if it cannot load a symbol file with an ID that matches a library or module:

<system symbols missing>

<symbols missing for uuid: xxx...>

The Performance Reporting service missing symbol file generates reports that appear in the Unity Services Dashboard under the Performance tab in the Reports section.

Missing system symbols

Reports where the system symbols are missing contain a line in the stack trace located in the Reports section of the Dashboard:

<symbols missing for uuid: xxx...>

Typically, the issue is that Unity does not have the symbols for that version of the operating system. For **iOS** and other Apple platforms, it can be difficult to obtain the symbols for older versions of the operating system. Unity has system symbol coverage on about 80% of **iOS** versions, starting from the **iOS** 7 family. Unity makes an effort to capture symbols for new **iOS** versions as soon as possible after they are released.

If you encounter this situation, you can check to see if you have similar crash reports for another version of the operating system. If you do, you might be able to resolve the issue by debugging it in that version of the operating system.

Missing application symbols

Reports where the application symbols are missing contain a line in the stack trace:

<symbols missing for uuid: xxx...>

When you build a project on which Performance Reporting is enabled, Unity generates a symbol file and uploads it to the Performance Reporting servers. If the process fails, the symbols missing message appears in the Services Dashboard.

To troubleshoot failures in the symbol upload, check the `symbol_upload.log` file located in the same folder as the main Unity log. It should indicate which symbols were found and processed, along with any errors that occurred during processing and uploading of the symbols.

Finding and uploading missing iOS/OSX application symbols

Leave feedback

When you build your application using Xcode, **iOS**/OSX places the symbols in a dSYM folder with a name similar to the following:

~/Library/Developer/Xcode/DerivedData/<build id>/Build/Products/<build type>/appname.dSYM

A dSYM file is an ELF file that contains DWARF (debugging with attributed record formats) debug information for your application. DWARF is a debugging file format that supports source-level debugging.

To verify that the dSYM contains the correct UUID, run the `dwarfdump` command, and replace `appname` with the name of your application's dSYM:

```
dwarfdump -u appname.dSYM/Contents/Resources/DWARF/appname
```

Dwarfdump is an application that prints DWARF information in a human-readable format.

The following shows sample output from dwarfdump:

```
UUID: 5EEDCCD2-38E7-3E52-81EC-B90C7BCD6D91 (armv7) appname.dSYM/Contents/Res
UUID: 583173FD-6697-3E3C-90DC-EA9147563A5B (arm64) appname.dSYM/Contents/Res
```

Note: The output of dwarfdump reports the UUID in upper-case and with dashes. UUIDs are often displayed as all lower-case with no dashes. Either format can represent a UUID. For example, `5EEDCCD2-38E7-3E52-81EC-B90C7BCD6D91` and `5eedccd238e73e5281ecb90c7bcd6d91` represent the same UUID.

When you have located the correct dSYM folder, zip the entire folder and upload it to the Performance Reporting service using the **Manage Symbols** tab on the Services Dashboard.

iOS, Bitcode, and the App Store

When you build your iOS app with Bitcode enabled and submit it to the App Store, Apple post-processes your build and creates a new binary. The new binary has a new UUID and a new corresponding dSYM. In this case, you must download the dSYM from iTunes Connect. It's available in iTunes Connect at **My Apps** > **Activity** > **All Builds** > **(choose your build)**:

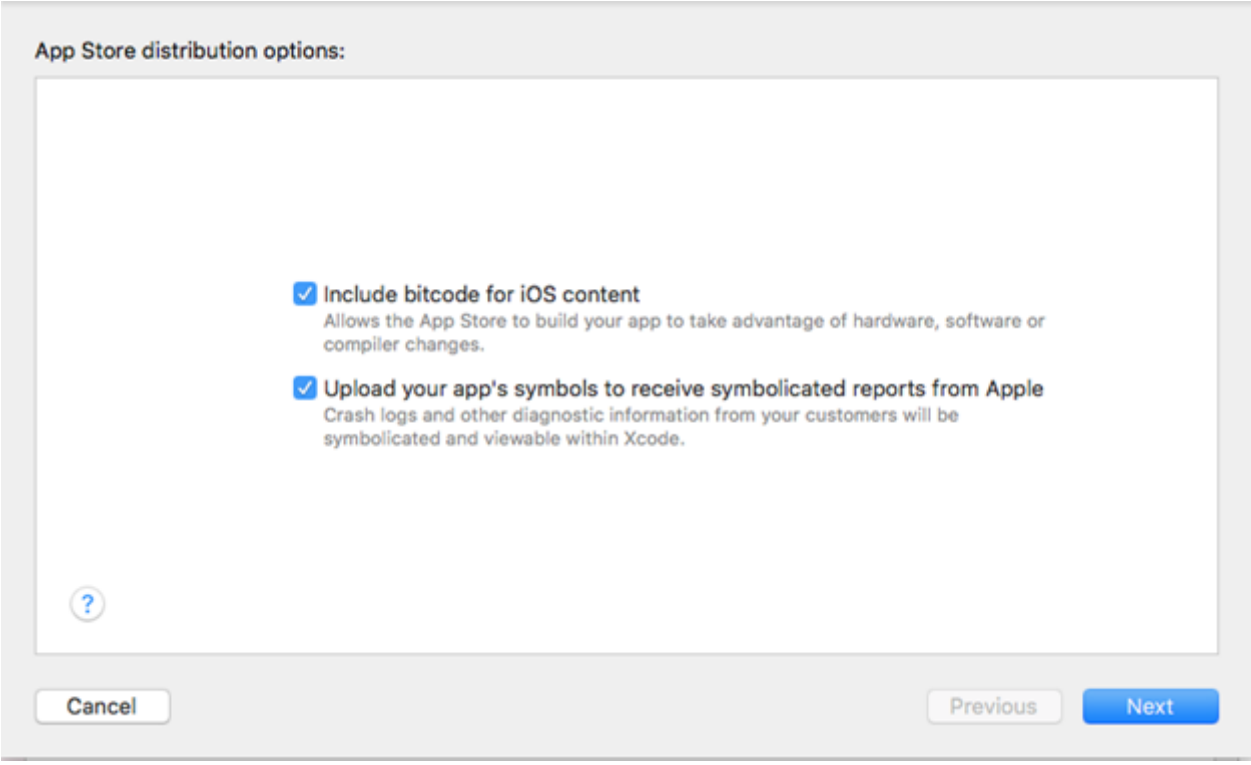
iOS Builds > 2017.10.20 (1)

General Information

App Name	Binary State	Bundle Short Version String
Performance Reporting Test	Validated	2017.10.20
Upload Date	**Build SDK**	**Bundle Version String**
Nov 20, 2017 at 9:43 AM	15B87	1
Original File Name	**Build Platform**	**Bundle ID**
Unity-iPhone.ipa	15B87	com.unity3d.performancereportingtest
Compressed File Size	**On Demand Resources**	**Includes Symbols**
24.2 MB | App Store File Sizes	No	Yes | Download dSYM
SiriKit		
No		

iOS Builds screen with “Includes Symbols” callout

When you upload your app to the App Store, check **Include bitcode for iOS content** and **Upload your app’s symbols to receive symbolicated reports from Apple** to enable the App Store to generate a new dSYM with the correct symbols:



App Store distribution options screen

When you have downloaded the new dSYM, verify that the UUID is correct using dwarfdump. Zip the file and upload it to the Performance Reporting service through the **Manage Symbols** tab.

Troubleshooting: I uploaded the symbols for the right UUID, but my reports are still not symbolicated correctly

When you upload new symbols through the **Manage Symbols** tab, there is a short period between the upload and when they are available for use. Please wait at least five minutes before submitting a new crash. When they

have been processed, any new reports are generated with the proper symbolication.

For more information on symbolication, see the Symbolicating Crash Reports section of Understanding and Analyzing Application Crash Reports on the Apple Developer website.

Note: When you upload a new dSYM lD, reports are not re-symbolicated. You must submit a new crash to generate a report that uses the new symbols.

Unity Integrations

Leave feedback

Unity Integrations allow you to connect the following Unity Services in your development workflow to non-Unity tools:

Bug Reports
Cloud Build
Collaborate
Performance Reporting

Impurtant: To add an integration, you must be an Owner or Manager on the Project. For information on assigning roles, see the Members & Groups section of Managing your Organization.

Unity Integrations supports the following non-Unity tools:

Webhooks: A user-defined callback that allows Unity to send POST requests to an external service.
Discord: Send notifications to your team's Discord channel through a Discord-defined webhook.
Slack: Send notifications to your team's Slack channel through a Slack-defined webhook.
JIRA (Bug Reports, Performance Reporting, and Collaborate only):
Bug Reports and Peformance Reporting: Create a JIRA ticket for a bug report.
Collaborate: Add comments to existing issues in JIRA. For example, you can add a comment to your issues any time you publish changes in Collaborate.

Adding an integration

To add an integration to your workflow:

Sign in to the Unity Services Dashboard.

Select the Project that you want to add an integration to.

In the left navigation column, click **Settings**, then click **Integrations (Beta)**.

Click the **NEW INTEGRATION** button.

In the **New Integration** dialog box:

Select the Unity Service to add an integration for, then click the **NEXT** button.
Select the event that triggers the action the integration takes, then click the **NEXT** button.
Select the program to integrate with, then click the **NEXT** button to configure the integration.
If you select a webhook or JIRA integration, the **Configure options** step opens. Otherwise, the configurations screen for the selected tool opens.

Webhook integrations

To configure a webhook integration, you must supply the following information:

Parameter	Description
Display Name	A name to identify the integration in the integrations list.
Webhook URL	The URL of the server endpoint that receives the webhook POST requests from the Unity service.

Parameter	Description
Authentication Secret	The client secret of your receiving application.
Content Type	The MIME type of the content. Select the content type of your data from the drop-down menu.
Disable SSL/TLS Verification	Set this option only when necessary.

Verifying SSL/TLS certificates helps ensure that your data is sent securely.

JIRA integration

The Unity JIRA integration allows you to interact with JIRA as follows:

Bug Reports and Performance Reporting: Create a new issue when you receive a report
Collaborate:Add a comment to an existing issue when you publish a change in Collaborate.

To configure a JIRA integration for Collaborate or Bug Reporting, you must supply the following information:

Parameter	Description
Display Name	A name to identify the integration in the integrations list.
JIRA Site URL	The URL of you JIRA instance.
JIRA Username	The user ID of an account that can post updates to your JIRA instance.
JIRA REST API Token	The API token to authenticate Integration requests to your JIRA server. For instructions on how to create a token, see Atlassian's documentation.

After you have entered the information for your JIRA instance, you must supply the following additional information for Bug Reports and Performance Reporting:

Parameter	Description
Create Issues In This Project	Select the project in which the issues are created.
Mark Incoming Issues As	Select the type of issue to log the report as.

When you make changes in Collaborate, update JIRA associated issues by referencing the issue key in your commit message. For example, "I fixed the crashes caused by ISS–42" adds publish details to issue "ISS–42".

Discord integrations

To configure Discord integrations, Unity calls an app that uses the Discord API to register a webhook to a Discord channel. If you do not have a Discord server, see Discord documentation: How do I create a server?.

To complete the configuration:

Sign in to your Discord account.
From the **Select a server** drop-down menu select your Discord server.
From the **Select a channel** menu select a channel to post notifications to.
Click the **Authorize** button.

Slack integrations

To configure Slack integrations, Unity calls an app that uses the Slack API to register a webhook to a Slack channel.If you do not have a Slack server, see Slack documentation: Create a Slack workspace.

To complete the configuration:

Sign in to your Slack account.
In the right-hand corner of the of the app, select a Slack workspace.
From the **Post to** drop-down menu, select the Slack channel to post notifications to.
Click the **Authorize** button.

Discord integrations

To configure Discord integrations, Unity calls an app that uses the Discord API to register a webhook to a Discord channel. If you do not have a Discord server, see Discord documentation: How do I create a server?.

To complete the configuration:

Sign in to your Discord account.

From the **Select a server** drop-down menu select your Discord server.

From the **Select a channel** menu select a channel to post notifications to.

Click the **Authorize** button.

Slack integrations

To configure Slack integrations, Unity calls an app that uses the Slack API to register a webhook to a Slack channel.If you do not have a Slack server, see Slack documentation: Create a Slack workspace.

To complete the configuration:

Sign in to your Slack account.
In the right-hand corner of the of the app, select a Slack workspace.
From the **Post to** drop-down menu, select the Slack channel to post notifications to.
Click the **Authorize** button.

Editing an integration

To edit an existing integration:

Sign in to the Unity Services Dashboard.
Select the Project that you want to edit an integration for.
In the left navigation column, click **Settings**, then click **Integrations (Beta)**.
Click **EDIT** next to the integration to modify.
The types of edits you can make depend on the integration. For Slack and Discord integrations, you can update the display name or delete the integration.

For webhook and JIRA integrations, you can modify any of the configuration parameters that you supplied when you created them.

2018–05–02 Page published with editorial review
2018–08–10 Page amended with editorial review

Multiplayer Services

Leave feedback

Unity's Multiplayer Service is the easiest way to set up real-time, networked games for Unity. It's fast to implement and highly customizable. Unity-provided servers and matchmaking services ensure that your players can easily find and play with each other.

To learn more about how to use the Unity Multiplayer Service, see the Services section of Multiplayer documentation.

XR

Leave feedback

XR is an umbrella term, encompassing **Virtual Reality** (VR), Augmented Reality (AR) and Mixed Reality (MR) applications.

This section provides overviews for the VR, **AR** and MR SDKs supported by Unity. Each overview contains a quick start guide, and advice on troubleshooting and best practice. You'll also find information on input and controller mappings, **XR rendering** and audio, and the Unity **XR** API.

XR Overview

Overview of XR concepts.

VR Overview

Overview of VR concepts.

Google VR

Learn about Google VR SDK support in Unity for Cardboard and Daydream devices.

Oculus

Learn about support in Unity for Oculus devices.

Open VR

Learn about OpenVR SDK support in Unity.

Windows Mixed Reality

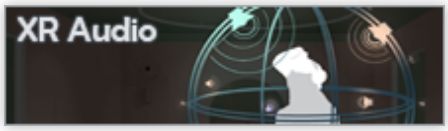
Learn about Windows Mixed Reality SDK support in Unity.

Vuforia

Learn about Vuforia SDK support in Unity.

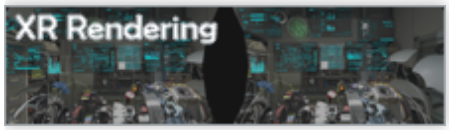
XR Input

Learn about the XR API input system and specific XR Platform interaction concepts.



XR Audio

Learn about Audio concepts in XR and how to implement Spatial Audio in your applications.



XR Rendering

Learn about XR Rendering concepts.

XR Reference

Learn about the XR API in Unity.

Further Resources

Unity Official XR forums
Unity Official XR tutorials
HTC Vive Developer forum
Official Oculus developer documentation
Windows Mixed Reality dev center documentation
Google VR Unity developer documentation and guides
Vuforia developer documentation and guides

XR SDKs

Leave feedback

This section provides overviews for all the **XR** SDKs supported by Unity, covering topics such as quick start guides, SDK configuration settings and troubleshooting.

Google VR

Leave feedback

Google VR is the technology behind both Google Daydream and Google Cardboard VR platforms. Google VR encompasses support for smartphones, head mounted viewers and controllers, standalone head mounted displays and applications. Google VR provides an SDK for Unity that allows you to build VR content for this rich ecosystem of devices.

Google VR technologies includes Daydream, a platform for high quality mobile VR and Cardboard, built for bite-size VR experiences and 360 video. Daydream consists of Daydream-ready phones, headsets, controllers and experiences and supports high-end Android mobile devices through Daydream. See the full list of Daydream devices on the Google VR) website. Cardboard works with almost any smartphone on both iOS and Android.

Google Cardboard is the most accessible mobile VR solution, allowing anyone with an **iOS** or Android smartphone to experience VR apps. Cardboard does have minimum and recommended requirements for its usage, but these are much less hardware-bound than Daydream, focusing mainly on the operating system and basic control systems (such as the gyroscope). Users can view VR apps with an official Google Cardboard viewing device, or a third-party VR viewing device which supports Google VR content (See Hardware and Software Requirements section). Cardboard also supports a range of third-party handheld controllers, for apps which require more complex interactions.

Daydream allows more feature-rich experiences than Cardboard, but only supports Daydream ready devices such as Daydream-ready phones, built for VR with high-resolution displays, smooth graphics and high fidelity sensors for precise head tracking. Daydream phone apps are accessible with the Daydream View, Google's headset and motion controller for experiencing high quality VR, using a Daydream-ready phone. Daydream will also support a Daydream Standalone HMD built by original design manufacturers. Daydream Standalone is an untethered head mounted display running on the Android OS including richer controller input and motion tracking.

See Google's VR documentation for more information about the requirements and functionality of Cardboard and Daydream.

2018–03–27 Page published with editorial review

Google VR documentation updated for Unity XR API in 2017.3

Google VR hardware and software requirements

Leave feedback

This section provides an overview of both the minimum and recommended hardware and software requirements for using the Google VR SDK.

Below are the minimum platform requirements for devices to support the Google VR SDK, along with some important considerations:

You must have Android API SDK version 21 (also known as “Lollipop”) installed.

To develop Google VR applications, you must have a Cardboard- or Daydream-supported phone.

See Google’s documentation on Daydream hardware to learn how to set your Android phone up for Daydream development.

Minimum hardware and software requirements

This section provides an overview of the minimum hardware and software requirements for developing with the Google VR SDK.

Device	Hardware	Software
Cardboard	- Devices running Android 4.1 or later with a gyroscope. - Devices running iOS 8 or later with a gyroscope. - A viewing device compatible with Cardboard.	- Unity Cardboard integration requires Android Lollipop or greater. - iOS devices with the Google Cardboard application installed.
Daydream	- Daydream compatible devices. See the Google Daydream Hardware page for a list of devices compatible with Daydream.	

Recommended hardware and software requirements

This section provides an overview of the recommended hardware and software requirements for developing with the Google VR SDK.

Device	Hardware	Software
Cardboard	- For Android, refer to the documentation provided with your Cardboard viewer for a full list of compatible devices. - For iOS, any iPhone 5 series or later phone. - A viewing device compatible with Cardboard.	- Android 5.0 or later. - iOS 8 or later, with the Google Cardboard application installed.
Daydream	- Any device compatible with the Daydream specification. See the Google Daydream Hardware page for a list of devices compatible with Daydream.	Android 7 or later.

You can find detailed information relating to Google VR development on the main Google developer site for VR. This site includes links to SDKs, instructional materials, and documentation on APIs and store publishing.

2018–03–27 Page published with editorial review

Google VR documentation updated for Unity XR API in 2017.3

Google VR quick start guide

Leave feedback

This section helps you get started with developing Google VR applications using Unity. Many of the instructions for developing Google VR content are available on the Google VR Developer website.

Use the following guides to install Unity and the Google VR SDK:

See Downloading and installing Unity to help install the Unity Editor.

i) When targeting Android, make sure you install Unity Android Support through the **Unity Installer**.

ii) When targeting iOS, make sure you install Unity iOS support through the **Unity Installer** and that you have a macOS machine to compile and deploy to your device.

Follow the instructions on the Google developer site for VR for your targeted platform(s).

Official Google VR guides

Google provides starter guides for developing Google VR applications with Unity for both Android and **iOS**. Links to specific guides are given below:

Getting started with Google VR SDK for Unity for Android.

Getting started with Google VR SDK for Unity for iOS.

Unity Platform configuration settings

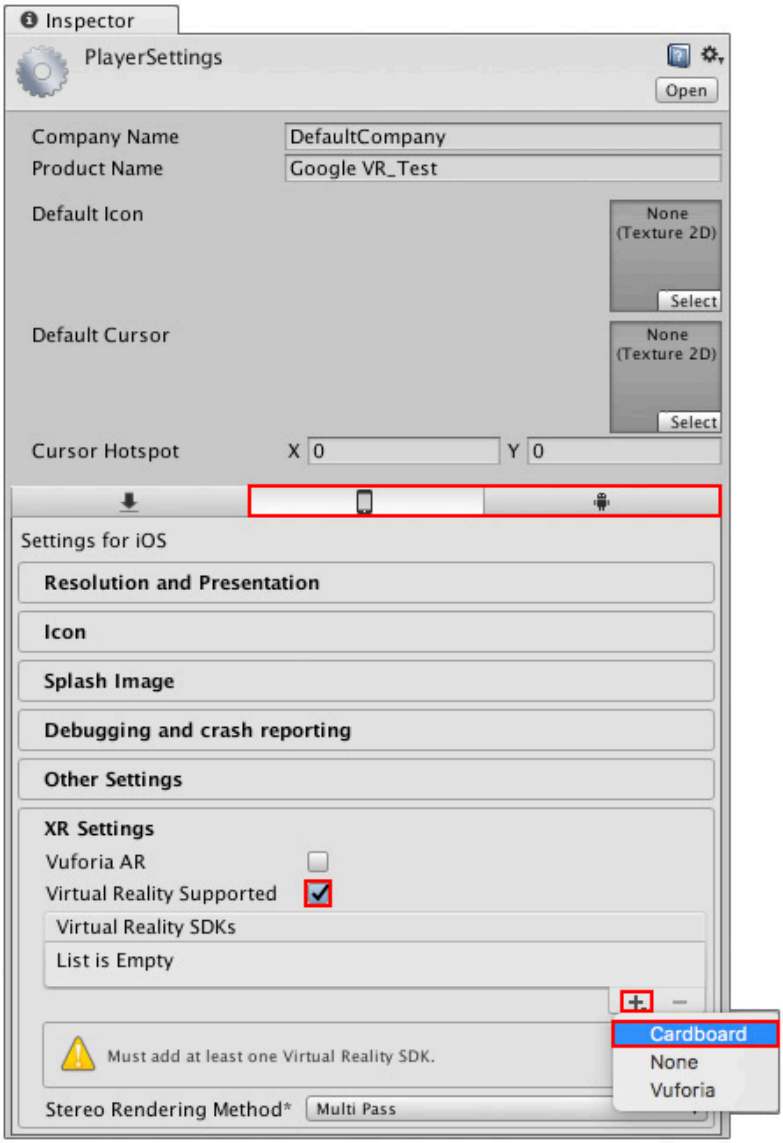
This section provides a step-by-step guide for configuring your Unity Project to build Google VR applications.

Follow the steps in the next sections to ensure your Unity project can successfully build for your target Google VR devices.

Targeting Cardboard (Android and iOS)

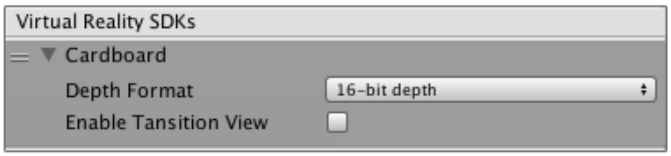
Unity provides a number of platform-specific build options for building applications to Cardboard-supported devices.

To access these settings, open the **Player Settings** in the Unity Editor (menu: **Edit** > **Project Settings** > **Player**) and click either the iPhone or Android icon, depending on which device you want to build to. Navigate to **XR Settings** and make sure **Virtual Reality Supported** is checked. Next, navigate to the **Virtual Reality SDKs** list and click the plus **(+)** button. Select **Cardboard** to add it to the list.



Finding Cardboard-specific settings for iOS/Android (Edit > Project Settings > Player > XR Settings)

When you have added **Cardboard** to the **Virtual Reality SDKs** list, click the foldout arrow next to it to view additional **Cardboard** settings.



Cardboard-specific settings in VR SDK list

The table below provides a list of properties available in the **Cardboard** settings, along with a description of their functionality.

Property	Description
Depth Format	Use this drop-down to set the Z buffer depth. Unity uses this to sort the visible data and determine what is actually rendered to the screen.
Enable Transition View	The transition view is the view that Google VR provides to inform the user that they must put their device in a Cardboard-compatible viewer. You can enable it to provide users some time to insert their device into a viewer. By default, this is disabled.

The minimum platform requirement for Cardboard is Android 5.0 (Lollipop) (SDK API level 21).

To ensure Unity uses the correct **APK**, and only runs on devices upgraded to the latest version of Android, you need to change the **Minimum API Level**.

To do this, go to the **Player Settings** and navigate to **Other Settings**. Under **Identification,** use the **Minimum API Level** drop-down to set it to the latest API in the list.

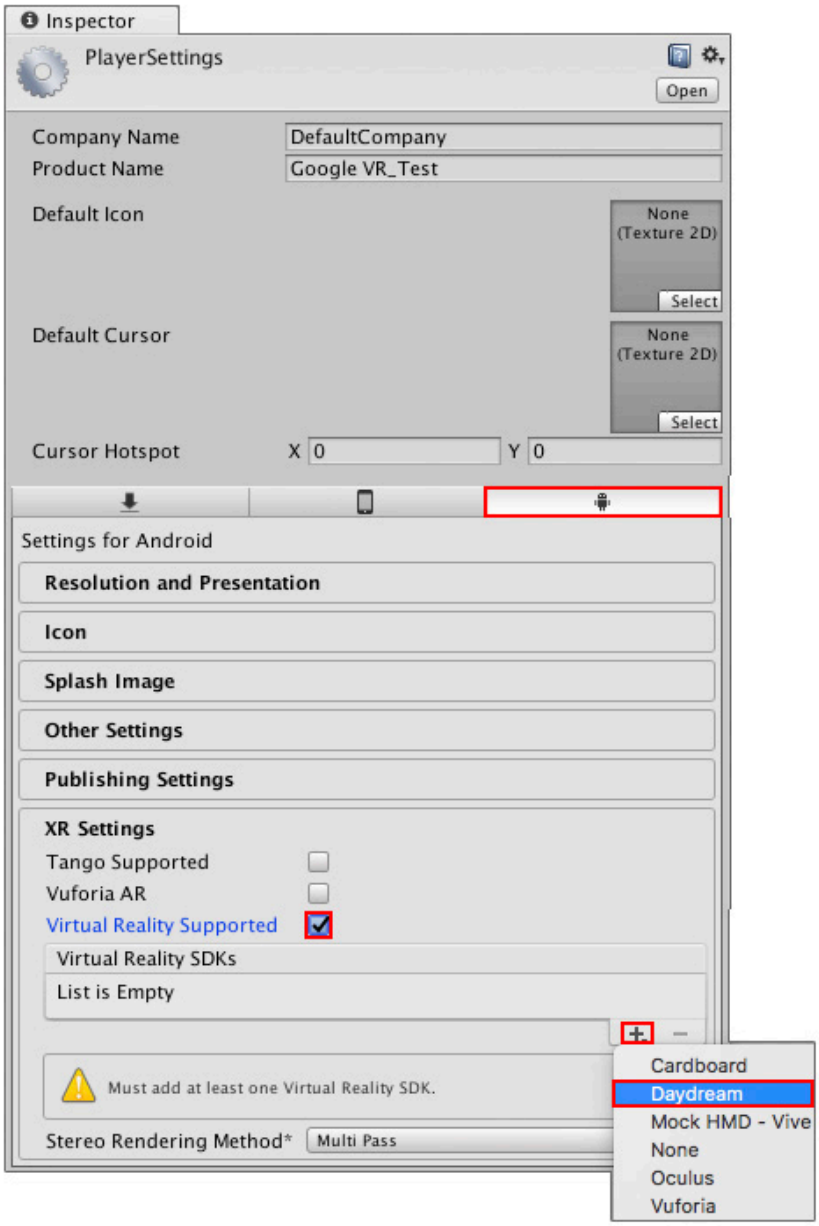
Set the **Target API Level** to API level 21 or higher for Cardboard only. If you are targeting Daydream then this needs to be 24 or higher. By default, this property uses the highest level you have installed. For more information about API levels for Android, see documentation on Android PlayerSettings.

You are now ready to create your Unity content for Cardboard. Follow the same workflow you would for normal Android or iOS development (see documentation on Android development or iOS development for more information). Make sure you build and run your game on a Cardboard-capable device (menu: **File** > **Build & Run**).

Targeting Daydream (Android only)

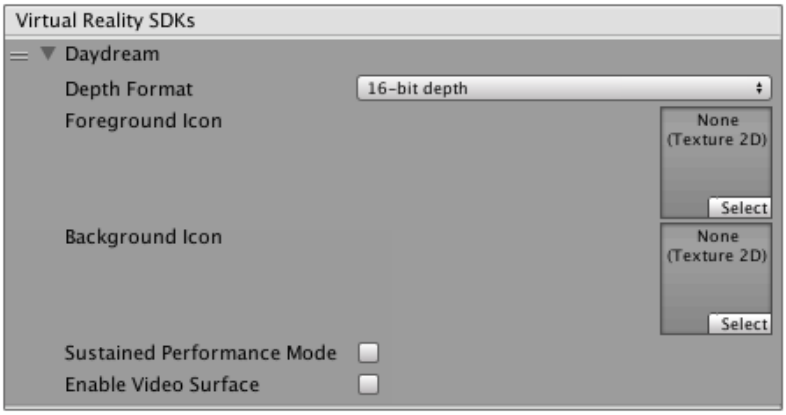
Unity provides a number of platform-specific build options for building applications to Daydream-supported devices.

To access these settings, open the **Player Settings** (menu: **Edit** > **Project Settings** > **Player**) and navigate to the Android section (click the Android icon marked in the figure below). Navigate to **XR Settings** and make sure the **Virtual Reality Supported** checkbox is ticked. Next, navigate to the **Virtual Reality SDKs** list and click the plus (+) button. Select **Daydream** to add it to the list.



Finding the Daydream-specific settings for Android (Edit > Project Settings > Player > XR Settings)

When you add **Daydream** to the **Virtual Reality SDKs** List, click the foldout arrow next to it to view additional Daydream settings.



Daydream-specific settings in VR SDK list

The table below provides a list of properties available in the Daydream settings, along with a description of their functionality.

Property	Description
Depth Format	Use this drop-down to set the Z buffer depth. This is used for sorting the visible data and determining what is actually rendered to the screen.
Foreground Icon	Set the foreground icon for presentation in the Google VR Play store.
Background Icon	Set the background icon for presentation in the Google VR Play store.
Use Sustained Performance Mode	Enable **Sustained Performance Mode** for longer **XR** experiences. This reduces performance to improve battery life.
Enable Video Surface	Enable Asynchronous Video Reprojection. For more information, see the documentation about Asynchronous Video Reprojection for more information.
Enable Protected Memory	Enables memory protection for DRM protected content when using Asynchronous Video Reprojection. This option only appears if **Enable Video Surface** has been selected. For more information, see Asynchronous Video Reprojection

The minimum platform requirement for targeting Daydream only is Android 7.0 (Nougat) (SDK API level 24). If you are targeting Cardboard as well as Daydream, then the minimum supported API level is 21.

To ensure Unity uses the correct APK, and only runs on devices upgraded to the latest version of Android, you need to change the **Minimum API Level**.

To do this in the Player Settings, navigate to **Other Settings**. Under **Identification,** use the **Minimum API Level** drop-down to set it to the latest API in the list.

Set **Target API Level** should be set to API level 24 if you are Daydream. By default, this property uses the highest level you have installed. For more information about API levels for Android, see documentation on Android PlayerSettings.

You are now ready to create your Unity content for Daydream. Follow the same workflow you would for normal Android development (see documentation on Android development for more information). Make sure you build and run your game on a Daydream-capable device (menu: **File** > **Build & Run**).

2018–03–27 Page published with editorial review

Google VR documentation updated for Unity XR API in 2017.3

Supported APIs and SDKs

Leave feedback

This section gives an overview of the supported APIs and SDKs for developing Google VR applications with Unity.

Unity API support

The following APIs are **not supported** by Google VR for Unity:

XRSettings.showDeviceView

XRDevice.isPresent

XRStats.gpuTimeLastFrame

Supported graphics APIs

The following **graphics APIs** are supported in Unity on both **Android** and **iOS** devices:

Android:

OpenGL

iOS:

Metal

OpenGL

2018–03–27 Page published with editorial review

Google VR documentation updated for Unity XR API in 2017.3

Google VR controllers and input devices

Leave feedback

Currently the only official firtst party input device for Daydream is the Daydream controller. **Google Cardboard** applications can use a variety of third party controllers to take input.

Daydream controller

Daydream controller

The Daydream controller allows 3 degrees of freedom, providing rotational and positional information, a dual axis touch/click controller along with 2 extra buttons.

For information on how to take input from this device see the Google Daydream Controller documentation.

2018–03–27 Page published with editorial review

Google VR documentation updated for Unity XR API in 2017.3

Google VR features overview

Leave feedback

For full details on specific features provided by Google VR, see Google's Daydream Developer documentation.

For an overview of features supported by the Google VR SDK for Unity, see Google's documentation on Google VR for Unity.

The next section provides a short overview of Google VR supported features.

Supported feature sets

The table below gives an overview of supported features for each target mobile VR device when working with Google VR in Unity.

Device	Platform	Rendering	Input
Cardboard	Android Lollipop or later	- OpenGL - Stereo Instanced Rendering - Stereo Rendering	
Cardboard	iOS	- OpenGL - Metal -Stereo Instanced Rendering -Stereo Rendering	
Daydream	Android Nougat or later using compatible hardware	- OpenGL - Stereo Instanced Rendering - Stereo Rendering	Daydream controller

2018–03–27 Page published with editorial review

Google VR documentation updated for Unity XR API in 2017.3

Unity Google VR Video Async Reprojection

Leave feedback

What is Video Async Reprojection?

Async Reprojection Video is a layer (referred to as an **external surface**) that an application can use to feed video frames directly into the async reprojection system. The main advantages to using the API are:

Without the API, the video is sampled once to render it into the app's color buffer, then the color buffer is sampled again to perform distortion correction. This introduces double sampling artifacts. The external surface passes video directly to the EDS compositor, so it's only sampled once, thus improving visual quality of the video.

When using the external surface API, video frame rate is decoupled from the app frame rate. The application can take 1 second to render a new frame and the only result is that the user will see black bars when they move their head, the video will keep playing normally. This should significantly reduce dropped video frames, and maintain AV sync

The application can mark that it wants to playback DRM video, and the API will create a protected path that shows protected video and maintains Async Reprojection frame rate.

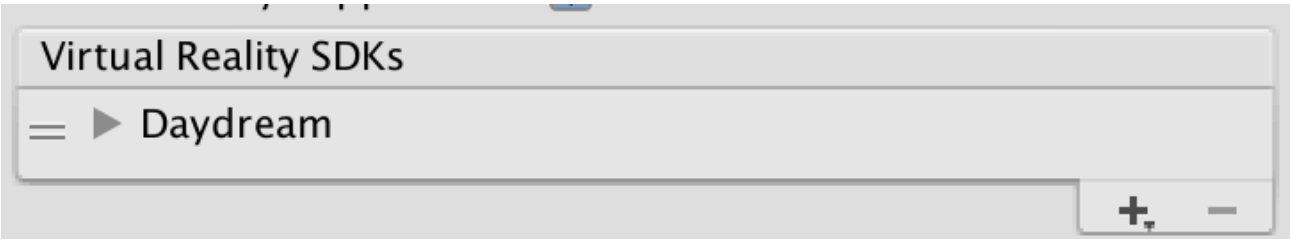
Known issues:

When using Video Async Reprojection, the **camera** must start at the origin (0,0,0). Errors may occur if the camera's position is not set to 0,0,0.

There is no publicly accessible C# interface for Async Reprojection. The public API is Java only.

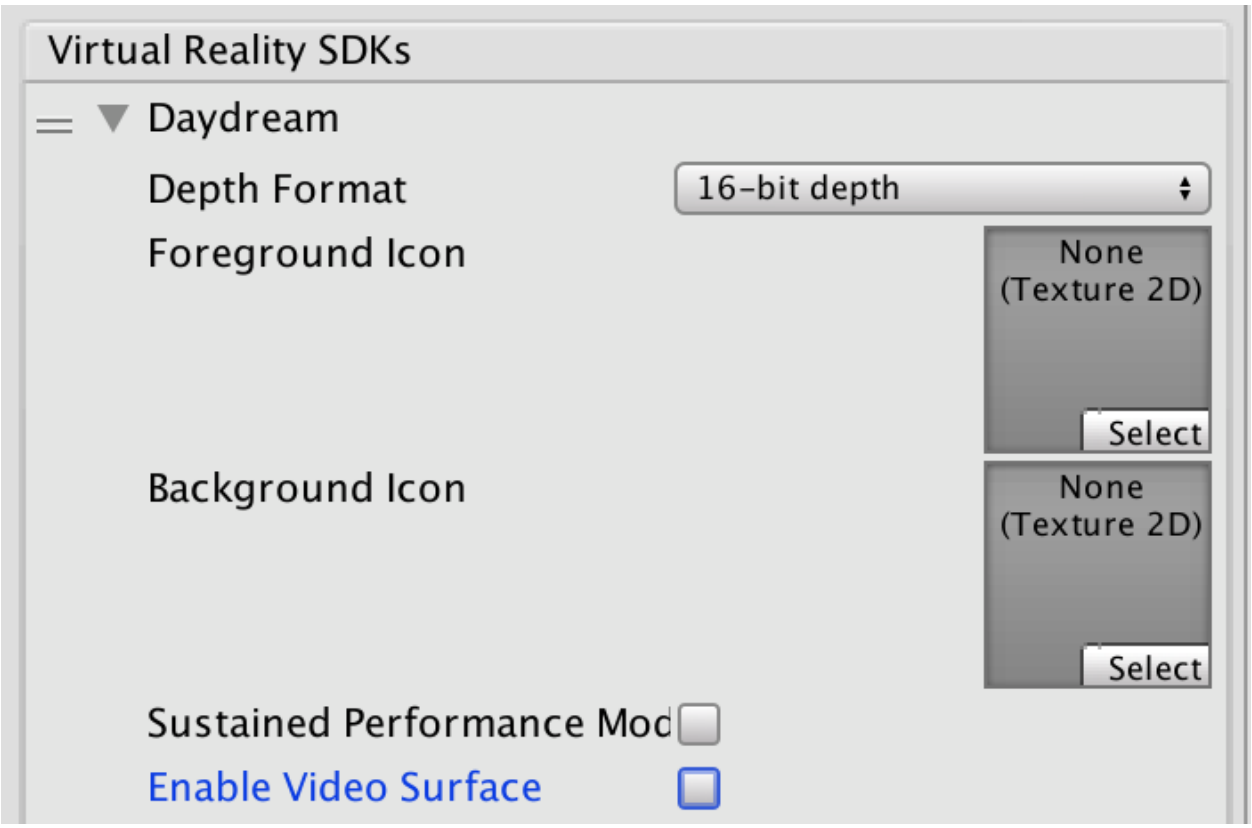
Enabling Async Video Reprojection

Async Video Reprojection is a part of the Daydream VR Device settings.



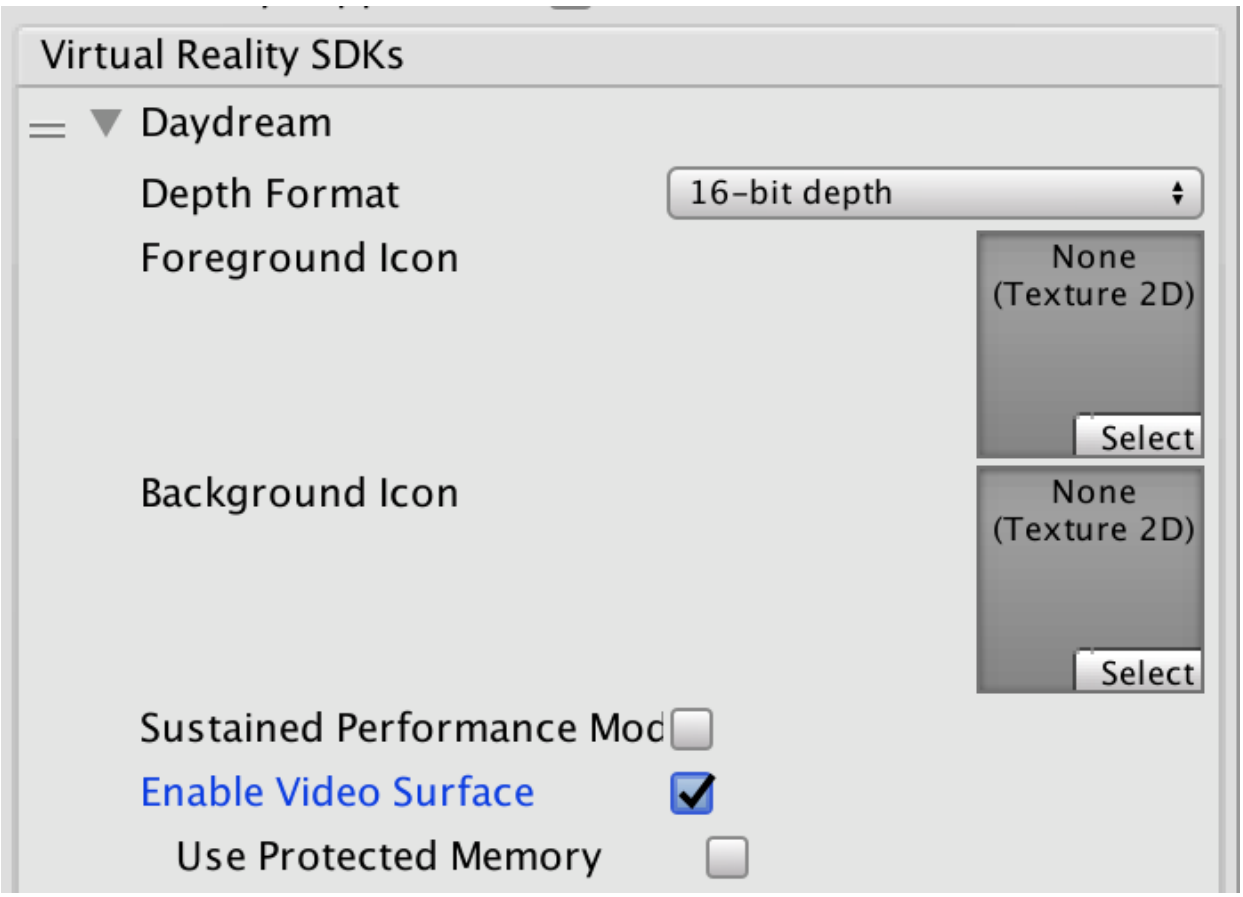
Adding Daydream to the Virtual Reality SDKs list settings

Click the grey arrow next to Daydream and then check the **Enable Video Surface** box to enable use of the Async Video Reprojection feature



Virtual Reality SDKs List Daydream settings

Select the **Use Protected Memory** option only if you require memory protection for all of your content as enabling this means that it is enabled for the lifetime of the application.



Enable Video Surface checkbox to enable Async Video Reprojection

API documentation

To take advantage of the Google VR API you will need to extend the UnityPlayerActivity. For more information, see documentation on Extending the UnityPlayerActivity.

Java plug-ins cannot directly access Objects in your **scene**, so you will need to provide a simple API to your C# code that will allow you to pass a transform to the Java side and tell your Java code when to start **rendering**.

Note: This code is not complete. It contains no implementation of a video player as that is a client specific implementation detail. It also doesn't have any playback controls, which would have to be implemented as objects in the scene and actions on those objects would need to call into Java.

For information on using Java in Unity and extending the UnityPlayerActivity, see documentation on Android development in Unity.

For information about the Google Video Async Reprojection system, refer to the Android Developer Network documentation on Video Viewports.

Java sample code:

```
package com.unity3d.samplevideoplayer;

import com.unity3d.player.GoogleVrVideo;

import com.unity3d.player.GoogleVrApi;

import android.app.Activity;

import android.os.Bundle;

import android.util.Log;

import android.view.Surface;

public class GoogleAVRPlayer implements GoogleVrVideo.GoogleVrVideoCallbacks {

    private static final String TAG = GoogleAVRPlayer.class.getSimpleName();

    private MyOwnVideoPlayer videoPlayer;

    private boolean canPlayVideo = false;

    private boolean isSceneLoaded = false;

    // API you present to your C# code to handle initialization of your

    // video system.

    public void initVideoPlayer(UnityPlayerActivity activity) {

     // Initialize Video player and any other support you need…

     // Register this instance as the Google Vr Video Listener to get

     // lifetime and control callbacks.
```

```
     GoogleVrVideo gvrv = GoogleVrApi.getGoogleVrVideo();

     if (gvrv != null) gvrv.registerGoogleVrVideoListener(this);

}



// API you present to your C# code to start your video system

// playing a video.

public void play()

{

       if (canPlayVideo && videoPlayer != null && videoPlayer.isPaused())

        videoPlayer.play();

}

// API you present to your C# code to stop your video system

// playing a video

public void pause()

{

        if (canPlayVideo && videoPlayer != null && !videoPlayer.isPaused())

        videoPlayer.pause();

}

// Google Vr Video Listener

@Override

public void onSurfaceAvailable(Surface surface) {

 // Google Vr has a surface available for you to render into.

 // Use this surface with your video player as needed.

     if (videoPlayer != null){

        videoPlayer.setSurface(surface);

        canPlayVideo = true;
```

```
            if (isSceneLoaded)

            {

                videoPlayer.play();

            }

          }

    }

    @Override

    public void onSurfaceUnavailable() {

     // The Google Vr Video Surface is going away. You need to remove

     // it from anything you have holding it and stop your video player.

        if (videoPlayer != null){

            videoPlayer.pause();

            canPlayVideo = false;

          }

     }

    @Override

    public void onFrameAvailable() {

     // Handle Google Vr frame available callback

    }
}
```

Unity C# sample code:

```
using System;

using System.Collections;

using System.Collections.Generic;
```

```
using System.Text;

using UnityEngine;

public class GoogleVRVideo : MonoBehaviour {

 private AndroidJavaObject googleAvrPlayer = null;

 private AndroidJavaObject googleVrVideo = null;

 void Awake()

 {

    if (googleAvrPlayer == null)

    {

      googleAvrPlayer = new AndroidJavaObject("com.unity3d.samplevideoplayer.GoogleAVRPlaye

    }

    AndroidJavaObject googleVrApi = new AndroidJavaClass("com.unity3d.player.GoogleVrApi"),

    if (googleVrApi != null) googleVrVideo = googleVrApi.CallStatic<AndroidJavaObject>("get

 }

 void Start()

 {

  if (googleVrVideo != null)

  {

   // We need to tell Google VR the location of the video suface in

   // world space. Since there isn't a way to get at that info from

   // Java, we can do it here and then pass the calculated matrix

   // down to the api we expose on our UnityPlayerActivity subclass.

   Matrix4x4 wm = transform.localToWorldMatrix;

   wm = Camera.main.transform.parent.worldToLocalMatrix * wm;

   wm = wm * Matrix4x4.Scale(new Vector3(0.5f, 0.5f, 1));


   // Convert 4x4 Row Ordered matrix into a 16 element column ordered
```

```
    // flat array. The transposition is to make sure that the matrix is

    // in the order that Google uses and the we flatten it to make passing

    // over the JNI boundary easier. The complication being that you have to

    // then convert it back to an 4x4 matrix on the Java side.

    float[] matrix = new float[16];

    for (int i = 0; i < 4; i++)

    {

     for (int j = 0; j < 4; j++)

     {

      matrix[i * 4 + j] = wm[j,i];

     }

    }

    googleVrVideo.Call("setVideoLocationTransform", matrix);

   }



   if (googleAvrPlayer != null)

   {

     AndroidJavaClass jc = new AndroidJavaClass("com.unity3d.player.UnityPlayer");

     AndroidJavaObject jo = jc.GetStatic<AndroidJavaObject>("currentActivity");

     googleAvrPlayer.Call("initVideoPlayer", jo);

     googleAvrPlayer.Call("play");

   }

  }

 }
```

2017–08–03 Page published with limited editorial review
Added in 2017.2

Google VR resources and troubleshooting

Leave feedback

Useful links

Below is a list of useful external links for further reading into Google VR development topics.

VR section on Google developer website

Cocoapods

Troubleshooting guide

This section provides a list of the most common problems you may encounter when developing with the Google VR SDK. The table below lists some common problems along with suggested solutions.

Problem	Suggestion
I have a problem with GvrController/GvrViewer/GvrXXXX/Instant Preview. What do I do to resolve this?	These types are part of the Google VR SDK for Unity and are owned and supported by Google. While the general Unity community may be able to answer your questions, for any technical issues you should visit the Google VR SDK for Unity site on GitHUB.
When is the Daydream Keyboard going to be released?	The Daydream Keyboard for Google VR is a Google technology and will be released in some future version of the core Android system. Access and use of this technology is based solely on Google and the Google VR SDK for Unity.
I am trying to build my Cardboard for **iOS** project, but Xcode is reporting linker errors.	Google distributes the Cardboard Native Development Kit (NDK) for iOS through the Cocoapods library management system. Unity is integrated with a specific version of that Cardboard NDK from the Cocoapods manager and uses the NDK to create your XCode project. The resulting project is generated differently from a standard Unity project. Cocoapods creates a wrapping XCode workspace containing the Unity project as well as a project for the Cardboard NDK library and its dependencies. Always make sure that you open and/or use the workspace and not just the project to avoid getting linker errors due to the missing libraries from Cocoapods.

See the official Google VR developer website for more troubleshooting information for both Cardboard and Daydream.

2018–03–27 Page published with editorial review

Google VR documentation updated for Unity XR API in 2017.3

Vuforia

Leave feedback

Vuforia is a cross-platform Augmented Reality (AR) and Mixed Reality (MR) application development platform, with robust tracking and performance on a variety of hardware (including mobile devices and mixed reality Head Mounted Displays (HMD) such as the Microsoft HoloLens). Unity's integration of Vuforia allows you to create vision apps and games for Android and **iOS** using a drag-and-drop authoring workflow. A Vuforia AR+VR samples package is available on the Unity **Asset Store**, with several useful examples demonstrating the most important features of the platform.

Vuforia supports many third-party devices (such as AR/MR glasses), and VR devices with back-facing **cameras** (such as the Gear VR). See the Vuforia page on Devices for a full list of supported devices. See the Vuforia API reference for more information about classes, properties and functions used in the SDK.

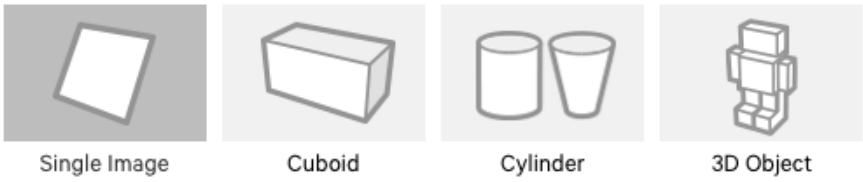
You can use any device with a camera to test **AR**/MR games and applications built in Unity with Vuforia.

Important concepts

Before learning more about Vuforia and its supported features, you need to understand a number of important concepts. Among these concepts are forms of tracking and marker types commonly used in Vuforia applications.

Marker-based tracking

In AR or MR, markers are images or objects registered with the application which act as information triggers in your application. When your device's camera recognizes these markers in the real world (while running an AR or MR application), this triggers the display of virtual content over the world position of the marker in the camera view. Marker-based tracking can use a variety of different marker types, including QR codes, physical reflective markers, Image Targets and 2D tags. The simplest and most common type of marker in game applications is an Image Target.



Common Image Target types

Image Targets

Image Targets are a specific type of marker used in Marker-based tracking. They are images you manually register with the application, and act as triggers that display virtual content. For Image Targets, use images containing distinct shapes with complex outlines. This makes it easier for image recognition and tracking algorithms to recognize them.

Markerless tracking

Applications using Markerless tracking are more commonly location-based or position-based Augmented or Mixed Reality. This form of tracking relies on technologies such as GPS, accelerometer, gyroscope and more complex image processing algorithms, to place virtual objects or information in the environment. The VR hardware and software then treats these objects as if they are anchored or connected to specific real-world locations or objects.

2018–03–28 Page published with editorial review

Vuforia documentation updated for Unity XR API in 2017.3

Vuforia hardware and software requirements

Leave feedback

This page provides an overview of both hardware and software requirements for using the Vuforia SDK with Unity.

Mobile devices

The table below lists the supported mobile device operating system (OS) versions required for running applications created with the Vuforia SDK. The table also includes supported OS and Unity versions for developing applications with Vuforia.

Device OS		Development OS		Unity Version	
Android (1)	4.1.x+	Windows (2)	7+	Windows (2)	2017.2+
iOS (2)	9+	OS X	10.11+	OS X	2017.2+
Windows (2)	10 UWP				

32-bit only

32 & 64-bit

Optical see-through digital eyewear

The Vuforia SDK supports a range of optical see-through digital eyewear devices. The table below lists supported devices, along with their required operating system (OS) versions. The table also includes supported OS and Unity versions for developing applications with Vuforia for these devices.

Device		Device OS		Development OS		Unity Version	
HoloLens	Current version	Android (1)(2)	4.0.3+	Windows (3)	7+	Windows (3)	2017.2+
ODG	R7+	Windows (1)	10 UWP	OS X	10.11+	OS X	2017.2
Epson	BT–200						

32-bit only

4.0.3 Native only support for Epson BT–200

32 & 64-bit

Vuforia tools

Some Vuforia tools are only supported on specific hardware. The table below lists these tools, along with their supported devices and required OS versions.

App	Devices	OS Version
Calibration Assistant	Moverio BT–200 ODG R–7	Android 4.0.3+
Object Scanner	Samsung Galaxy S8+ Samsung Galaxy S8 Samsung Galaxy S7 Samsung Galaxy S6	Latest supported OS on the device

For more detailed information on these tools, see the Vuforia Calibration Assistant and Object Scanner pages in the Vuforia developer library.

Virtual Reality SDK integration support

Below is a list of VR SDKs fully integrated into the Unity Editor, along with the release version of the Editor where they were first integrated.

VR SDK	Versions
Google VR SDK	Unity 2017.2
Cardboard Android SDK	Unity 2017.2
Windows Mixed Reality (HoloLens only)	Unity 2017.2

Graphics API support

The table below lists all the Graphics APIs supported by Vuforia when developing applications on supported operating systems.

Android	iOS	Windows
OpenGL ES 2.0 OpenGL ES 3.x	OpenGL ES 2.0 OpenGL ES 3.x Metal (iOS 8+)	DirectX 11 on Windows 10

For more information on using Vuforia with digital eyewear (such as the ODG R7+ and Epson BT–200), see Vuforia documentation on Vuforia for digital eyewear

For more information on using Vuforia with HoloLens, see Vuforia documentation on working with the Hololens sample in Unity

For more information on using Vuforia with the Google VR SDK, see Vuforia documentation on Developing for Google Cardboard

---

2018–03–28 Page published with editorial review

Vuforia documentation updated for Unity XR API in 2017.3

Vuforia quick start guide

Leave feedback

This guide provides instructions on setting up a Unity project with Vuforia and creating your first Vuforia application with Unity.

Contents

Setting up your Project for Vuforia
Setting up Image Targets
Importing and activating the Target Database in Unity
Adding Image Targets to your Scene
Displaying 3D models on top of tracked images

2018–03–28 Page published with editorial review

Vuforia documentation updated for Unity XR API in 2017.3

Setting up your Project for Vuforia

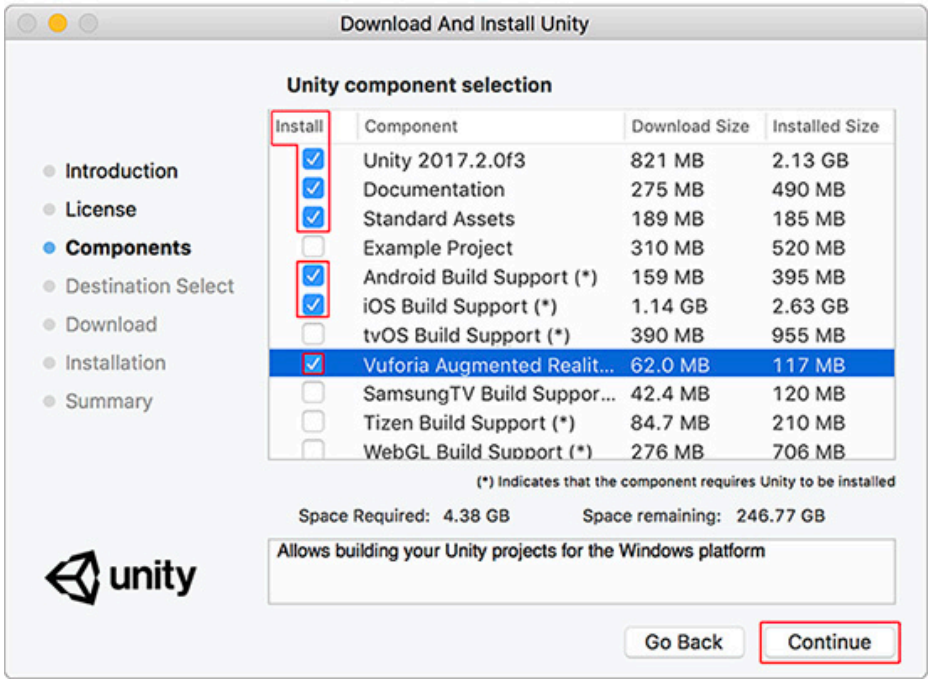
Leave feedback

Setting up your project to develop Vuforia **AR** or MR mobile applications is very similar to the set-up process for building with Unity for a mobile platform. The Unity Installer includes the Vuforia SDK. Follow the instructions for downloading and installing Unity on the InstallingUnity manual page. Vuforia provides a selection of **Prefabs** designed to be dropped into a **Scene** to provide feature functionality to your application. These are all available from inside the Unity Editor.

You should adhere to the same performance considerations as required for developing regular mobile games. For information on optimising for mobile devices, see Unity documentation on mobile optimisation.

To get Vuforia set up with Unity:

Install the latest version of Unity, and in the Unity **component** selection section of the installer, select **Vuforia Augmented Reality Support**, along with either the **iOS Build Support** or **Android Build Support** packages.



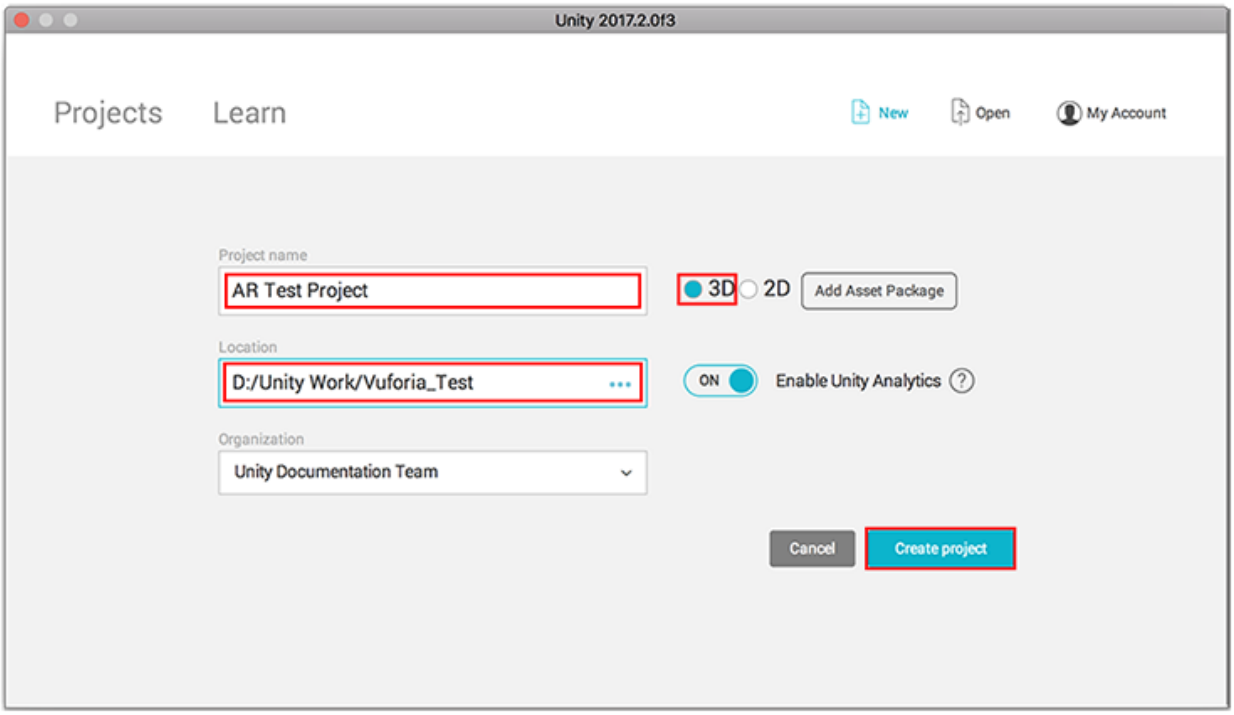
Installing Vuforia Augmented Reality support through Unity Installer

Note: Most AR and MR applications target mobile devices, so the focus of this guide is development for Android and iOS. See guides for enabling build support for Android and iOS devices in the Getting Started documentation for Android and iOS.

Create a Vuforia developer account from the Vuforia registration page. This account gives you access to the tools you need to make AR and MR applications with Vuforia in Unity.

If you have not already created a Unity ID, then do so from the Unity registration page. You need a Unity ID to download any packages from the Unity **Asset Store**.

Open Unity and create a new 3D Project (make sure the **3D** option is selected next to the **Add Asset Package** button). Name your Project, then click the **Create project** button.

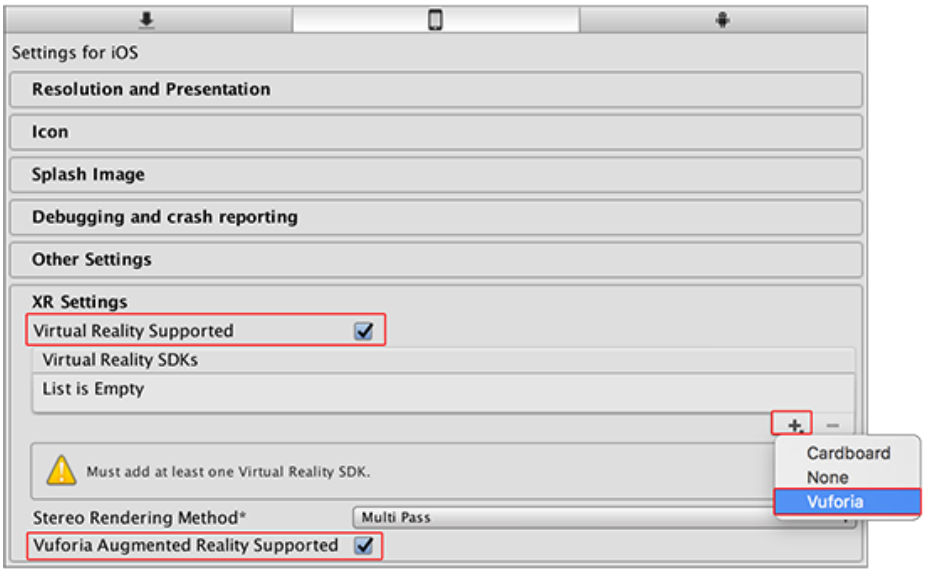


Creating a new 3D project

Tip: Download the Vuforia AR+VR Sample package from the Unity **Asset** Store. This package provides useful example Scenes demonstrating important features. You will not need this for following this guide, but it is useful to have for further learning later.

Activating Vuforia in Unity

To activate Vuforia in your Unity project, access the Player Settings from **Edit** > **Project Settings** > **Player** and select the tab for the mobile device you are building to. Under **XR Settings**, tick the **Vuforia Augmented Reality Support** checkbox.



Enabling Vuforia Augmented Reality Support in Unity Player Settings

Your Scene now contains two GameObjects: a main **Camera**, and a directional light. You need to add a new AR **Camera** to the Scene in order to enable AR functionality, and you need to delete the current **Main Camera** GameObject from the Scene.

To delete the Camera GameObject, select it in the **Hierarchy window** and press the delete key on your keyboard, or right click it and select **Delete**.

For more information on the individual settings in the **XR** Settings window, see the Vuforia Platform Configuration page of this manual.

Adding a Vuforia AR Camera and other GameObjects

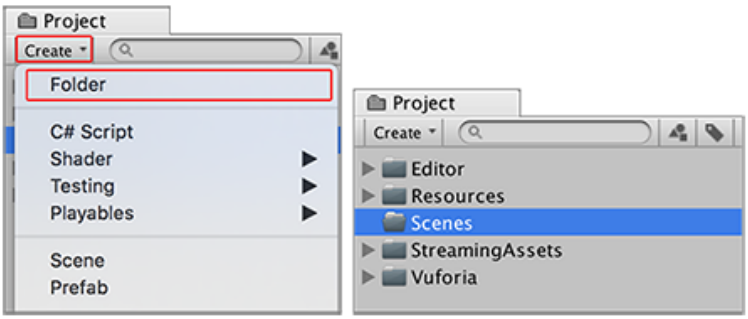
To add an AR Camera to your scene, go to **GameObject** > **Vuforia** > **AR Camera**.

If this is the first Vuforia GameObject added to your Scene then Unity also prompts to import your Vuforia Assets. Select **Import**, and Unity imports all necessary Vuforia files into your project.

The **Project window** displays 4 new folders, one of which is called **Vuforia**. The code and Assets in this folder provide the main AR and MR functionality. The other folders provide sample Scenes, resources, tools, and **plugins** so that you can develop AR and MR applications for a variety of devices.

Empty project with all imported Vuforia assets

Create a new folder in your project. To do this, navigate to the Project window, click the **Create** button, and select **Folder**. Name this new folder *Scenes* and save a new Scene inside this folder.



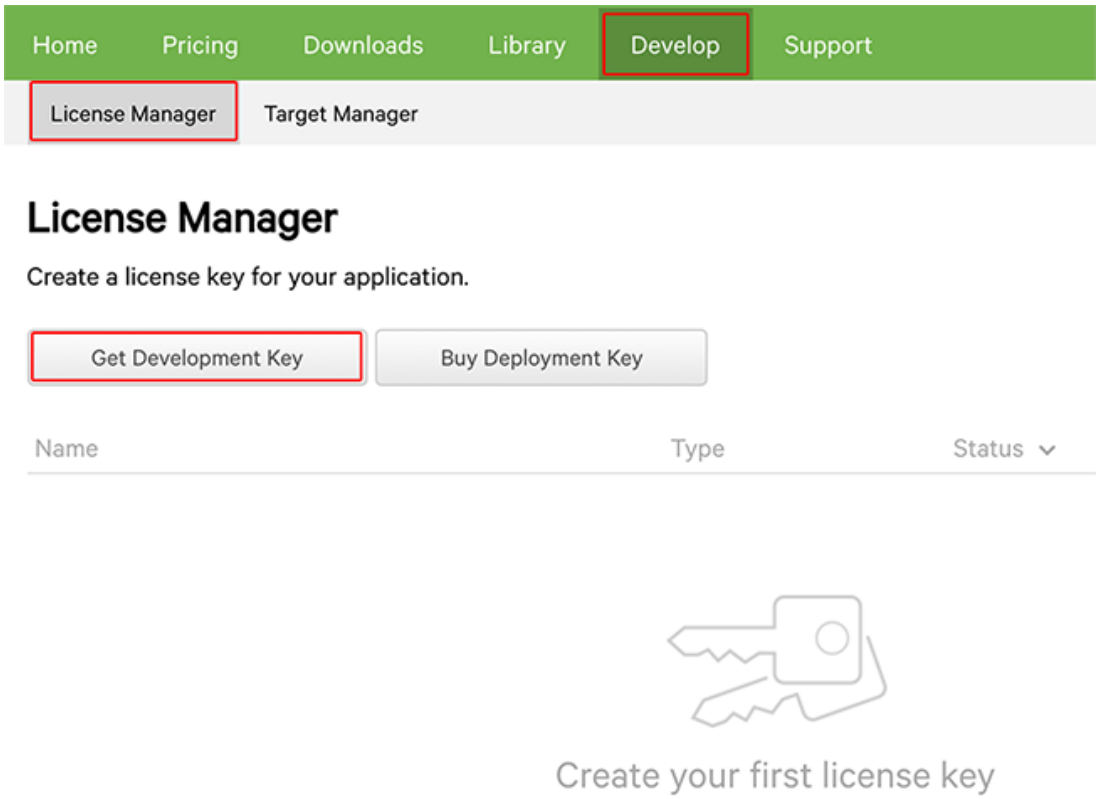
Creating an empty folder and naming it Scenes

This process also adds a new ARCamera GameObject in your Scene hierarchy.

Creating a Vuforia license key

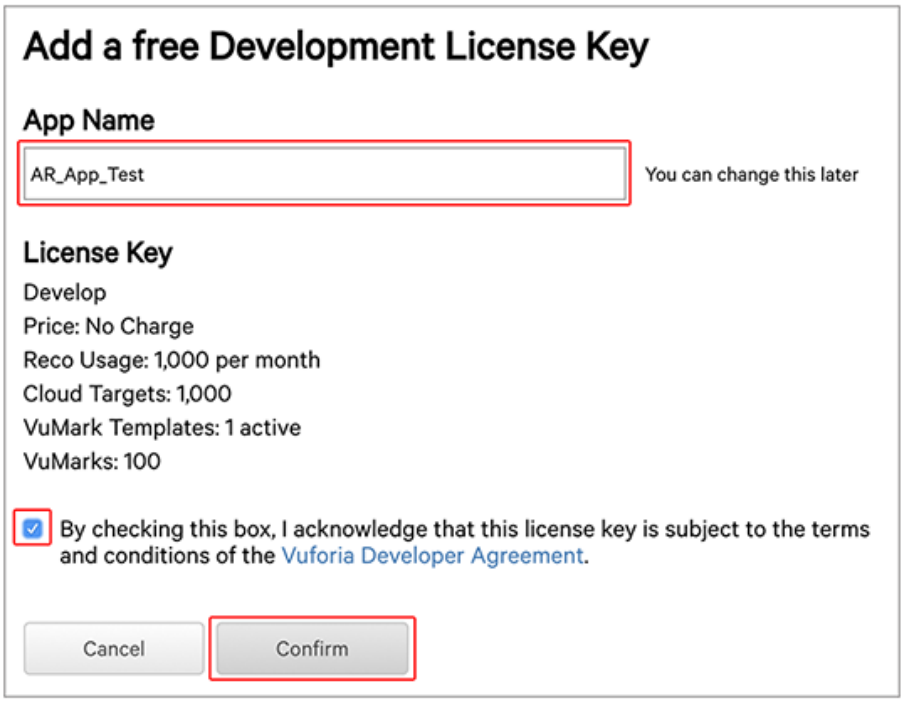
The last step in the setup process is to create a license key from the License Manager section of the Vuforia Developer Portal. You need to enter this into Unity's Vuforia configuration settings in order to build and test your application with Unity.

Visit the Vuforia Developer Portal and log in (or create a new account). Navigate to the **License Manager** in the **Develop** section and click the **Get Development Key** button to open the **Add License Key** page.



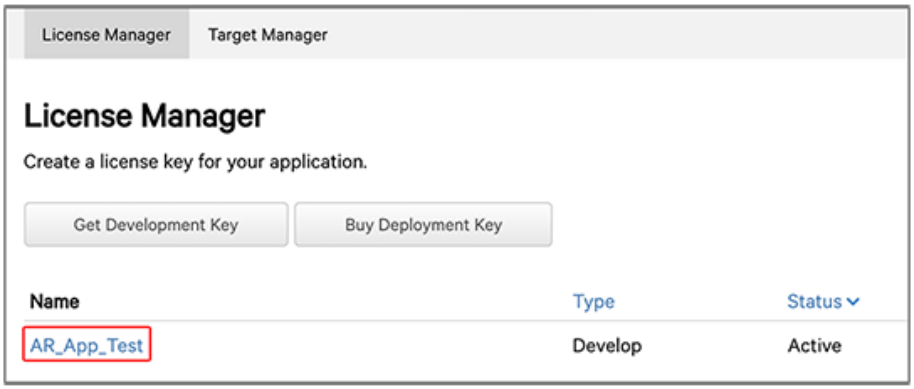
Creating a Vuforia Development Key

On the **Add License Key** page, enter a name for your app. Accept the terms and conditions, then click the **Confirm** button to generate a new license key.



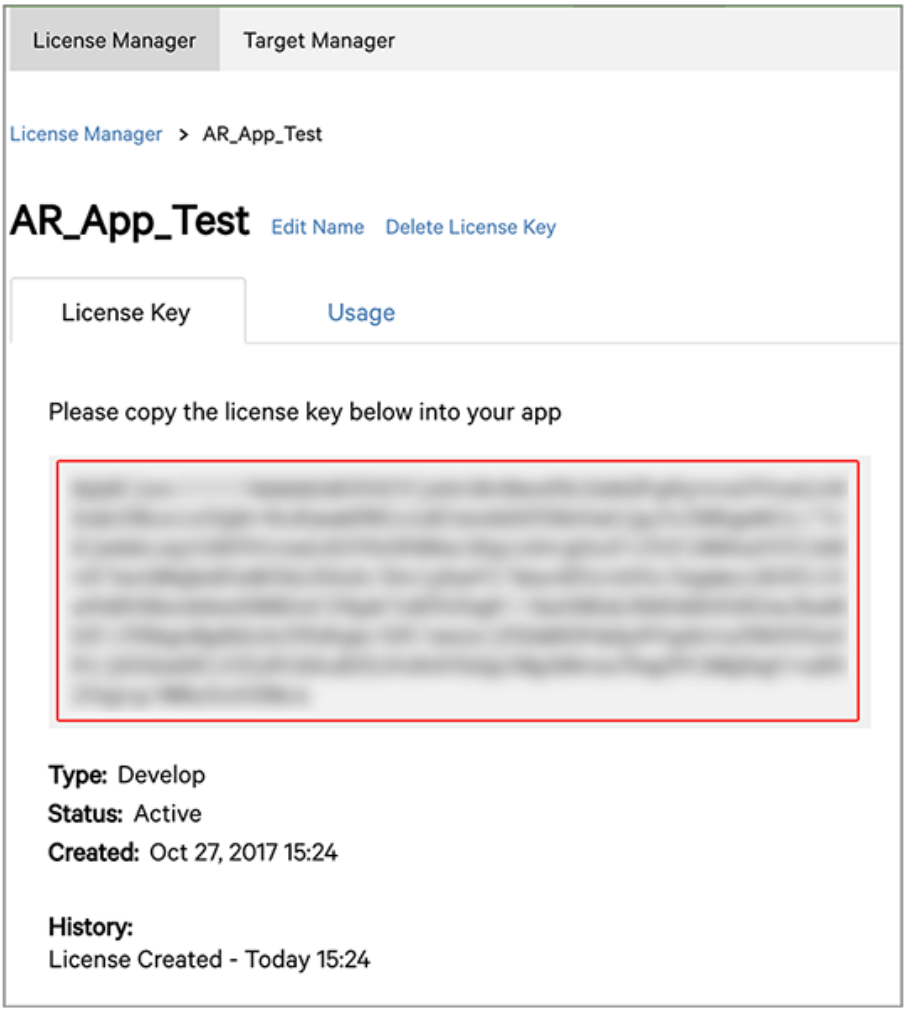
Confirming Vuforia development key details

On the next page, agree to the Vuforia Developer conditions (tick the box) and then click the **Confirm** button. This brings you back to the **Licence Manager** page where you can see your newly created license in a list where its status is **Active**. Click on the name of your App to view the license details. This allows you to retrieve your development license key.



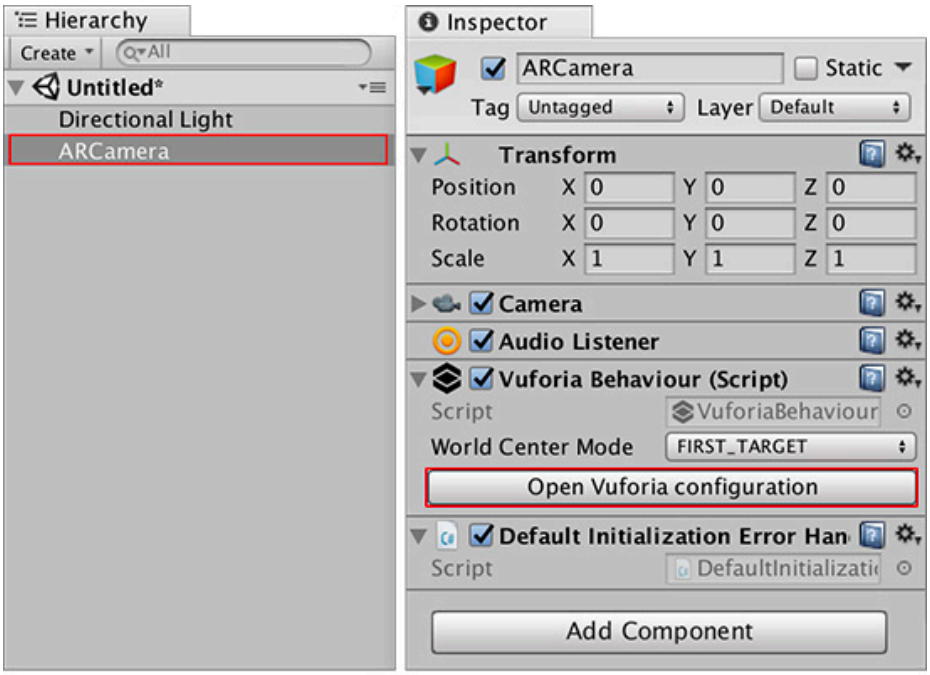
License Manager

Copy the license key to the clipboard and navigate back to your Unity Project.



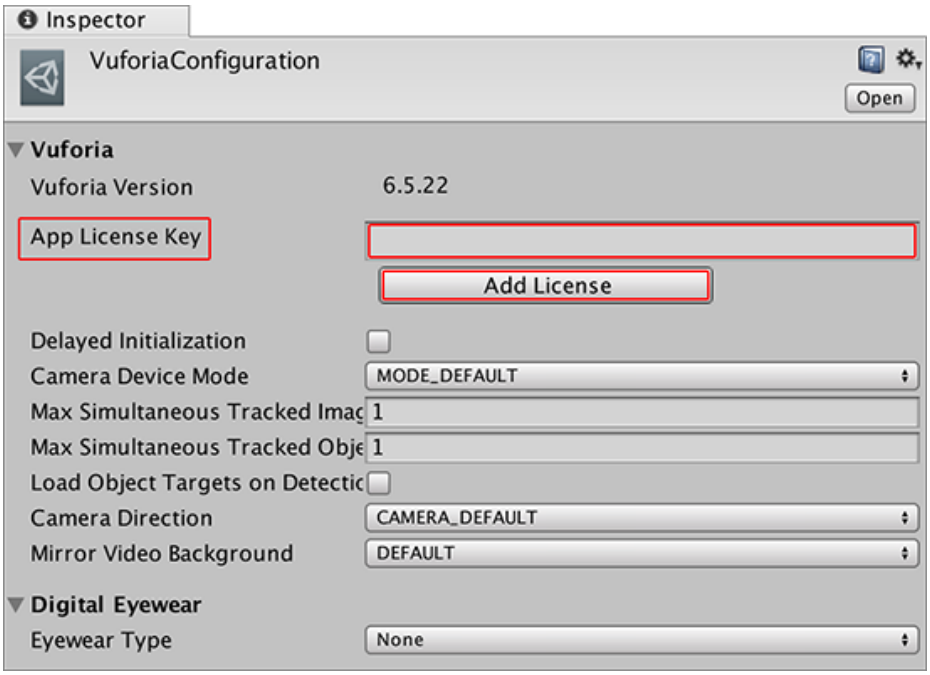
Copying your Vuforia License key

Select the **ARCamera** GameObject from the **Hierarchy window** and, in the **Inspector** window, navigate to the the **Vuforia Behaviour(Script)** component and click the **Open Vuforia configuration** button.



Accessing Vuforia configuration settings

The **Inspector window** displays a list of **Vuforia Configuration** options. Paste your Vuforia Development key into the **App License Key** text box under the Vuforia section and then click the **Add License** button.



Entering your Vuforia development key in the Vuforia Configuration settings

Testing your set-up

To test Vuforia apps in the Unity Editor, you need to have a webcam connected to your PC or laptop. As a final step to make sure Vuforia is now installed properly in your Unity Project, press the **Play** button to test your Scene. If Vuforia is set up correctly, a video feed from your webcam appears in the Editor Game View.

Now you are ready to set up Image Targets and add AR functionality to your Project.

2018–03–28 Page published with editorial review

Vuforia documentation updated for Unity XR API in 2017.3

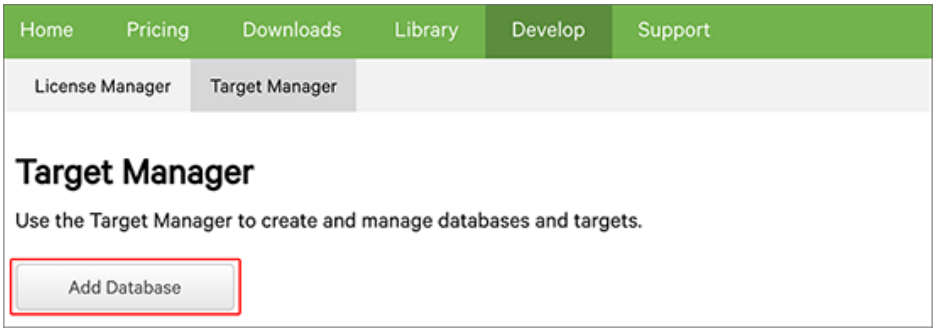
Setting up Image Targets

Leave feedback

This section shows you how to set up a simple Image Target and get it responding to basic tracking events.

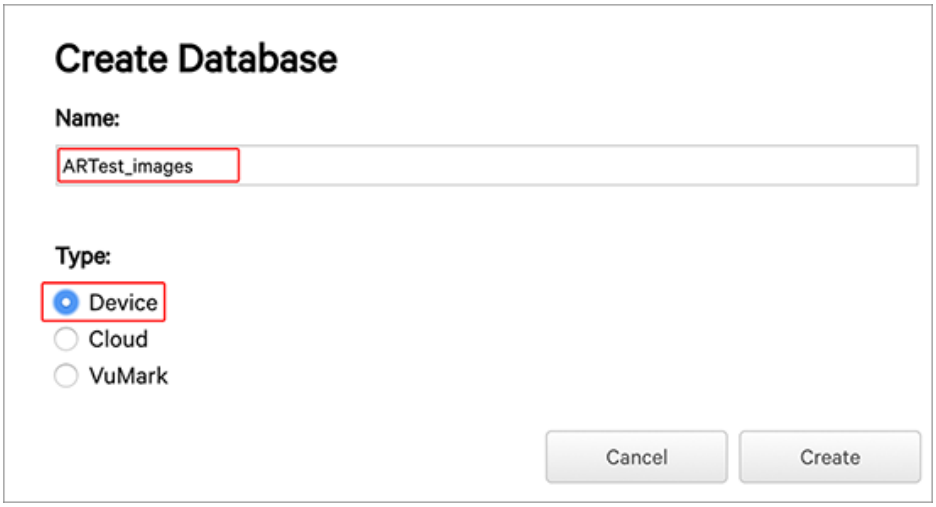
To allow your application to recognize images and use them as Targets to trigger gameplay, display graphics or information, you need to create a **Target database**. You can create Target databases directly from the Vuforia Developer Portal Target Manager page, as the steps in this section describe.

Log into your Vuforia Developer account. Then navigate to the Target Manager page and click on the **Add Database** button.



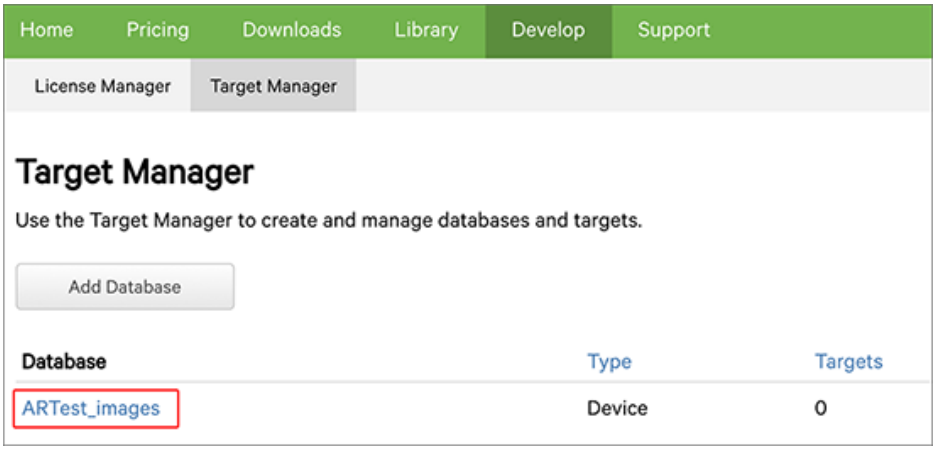
Adding a new Target Database

On the **Create Database** page, type a name for your database, select **Device** from the **Type** options, then click the **Create** button.



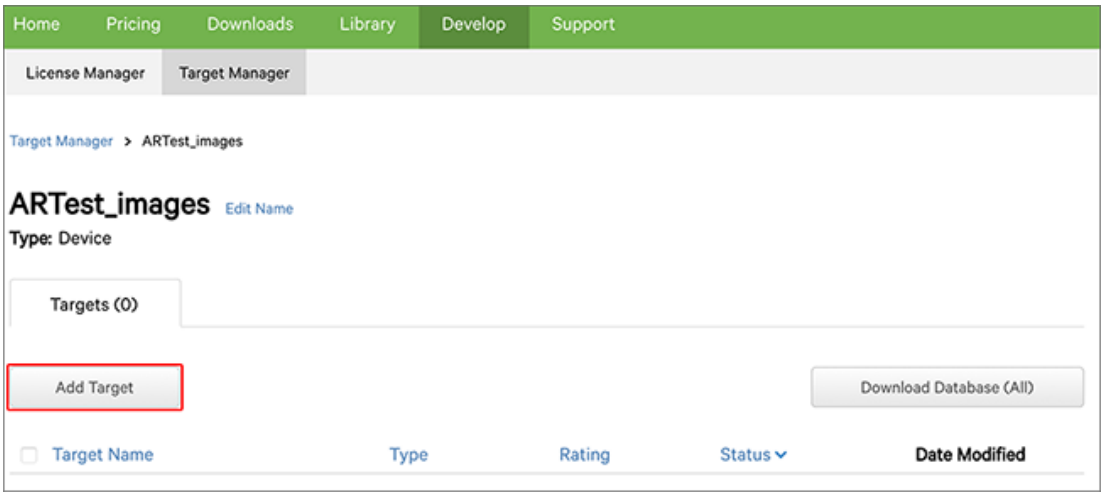
Creating a new Target Database

This adds the new Target database to the **Target Manager** list. Now click on the **Database name** in the list to open the **Device Database list**.



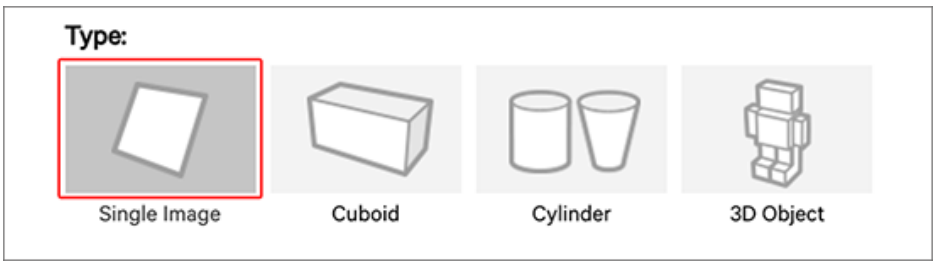
Managing the new Target Database

This brings you to the **Target list** page for the database, where you can add new Targets and download the database in specific formats for use with several platforms. Click the **Add Target** button to open the **Add Target** popup window.



Open Add Target window

The **Add Target** window presents you with options to specify details about the Target you want to add. There are four different types of Targets you can add: Single Image, Cuboid/Box, Cylinder and **3D Object**. Under **Type**, Select **Single Image** and browse your hard drive to locate the image you want to use as an Image Target.



Choosing Target Type

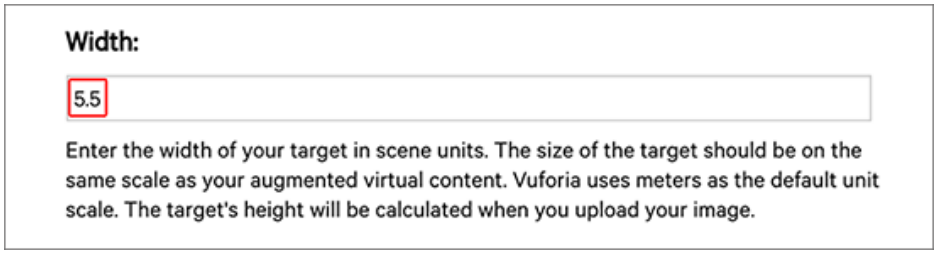
This example uses a playing card to demonstrate the Image Target recognition capabilities in Unity.

Image Target used

Use any image, but make sure that the image has enough detail to be rated as a 5-star Target so that the **camera** can easily track it.

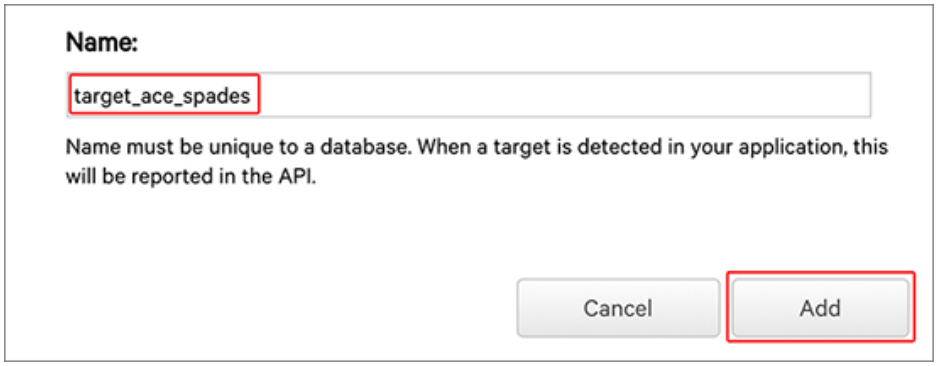
Choosing Target image

The **Width** value is a scale value that you need to set to the size you want the image to appear in your Unity **Scene** (in real-world units). Unity measures everything in your Scene in relation to the size of your Target image. For this example, the width of the playing card is 5.5cm, so you would use *5.5cm* as the **Width** value. If you need a larger size Target, then increase this **Width**.



Setting a Target Width

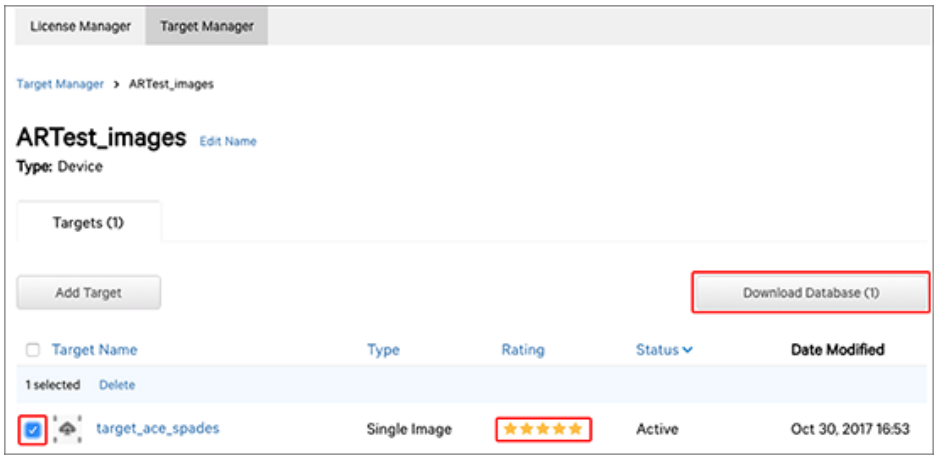
Enter a name for the Target image, and click the **Add** button to upload the Image Target to the database.



Naming the Target and adding it to the database

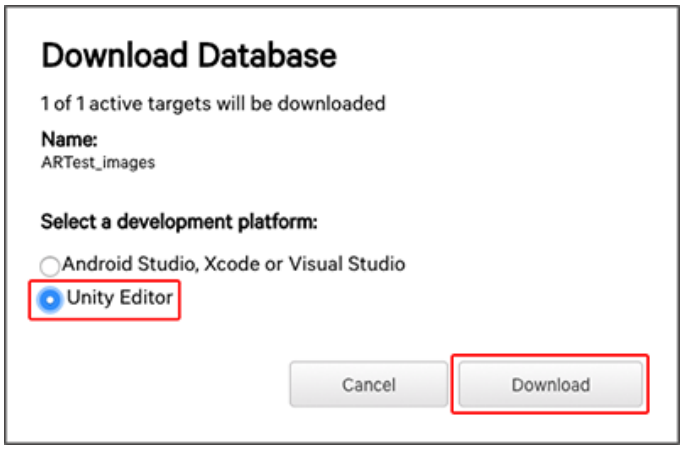
The image appears in the list of Targets with a **Rating** value represented by stars. If your **Rating** is less than 5 stars, it may be harder for the camera to track it. To learn more about what affects Image Target ratings, see Vuforia documentation on Optimizing target detection and tracking stability.

Once you are satisfied with your image's Rating, select the checkbox to the left of the **Image Target** name and click the **Download Database** button.



Downloading the Target database and Target quality rating

On the **Download Database** window, under **Select a development platform**, select **Unity Editor**, then click the **Download** button. This downloads a Unity package of the Target database that you can save on your hard drive.



Downloading Database Unity package

Switch back to your Unity project to import the Unity package for use in your application.

2018–03–28 Page published with editorial review

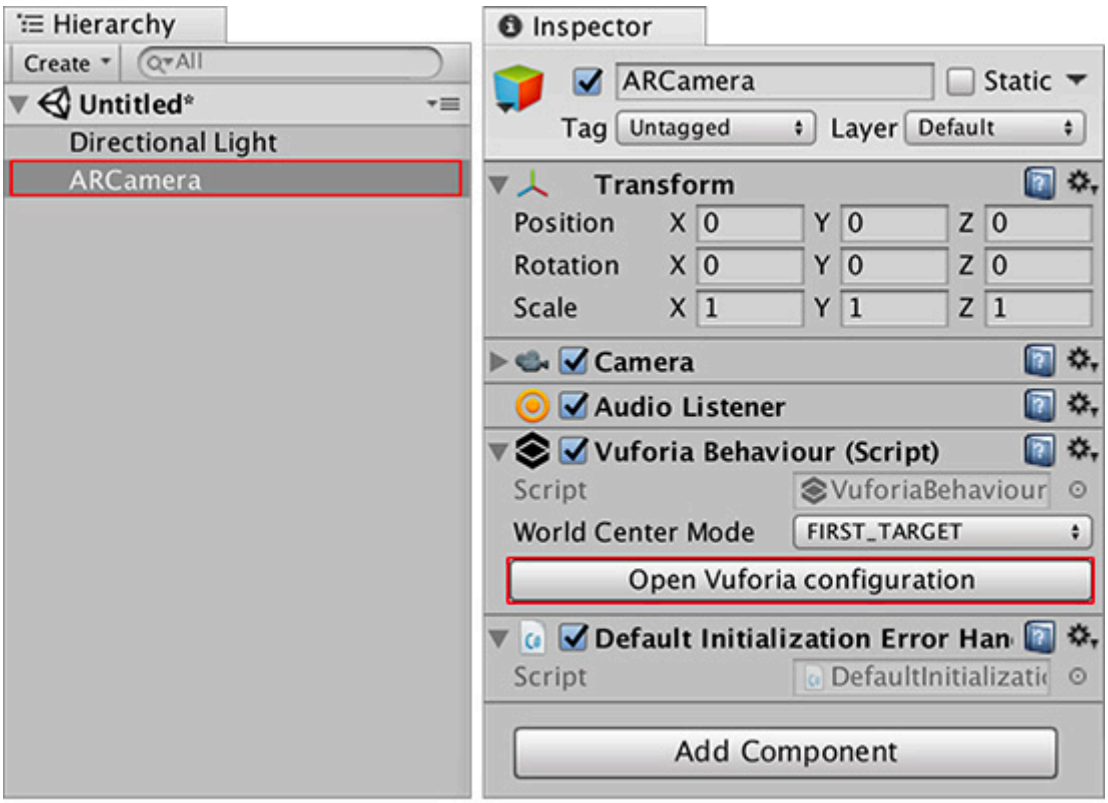
Vuforia documentation updated for Unity XR API in 2017.3

Importing and activating the Target Database in Unity

Leave feedback

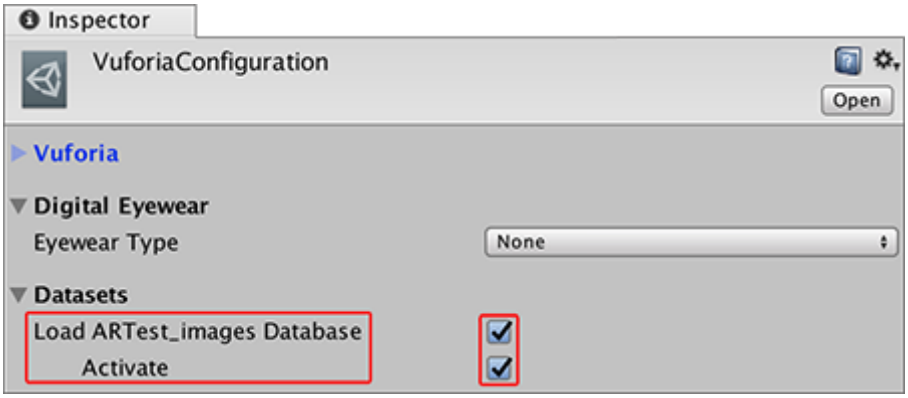
In the Unity Editor, navigate to **Asset** > **Import Package** > **Custom Package** and find the package on your hard drive. In the **Import Unity Package** window, click the **Import** button.

Select the **ARCamera** from the **Hierarchy window** and, in the **Inspector window**, navigate to the **Vuforia Behaviour (Script) component** and click on the **Open Vuforia configuration** button.



Accessing Vuforia configuration settings

In the **Vuforia Configuration** window, under **Datasets**, select and check both the **Load [DatabaseName] Database** and **Activate** checkboxes. This activates your Image Target database for use with Unity.



Activating imported Image Target database

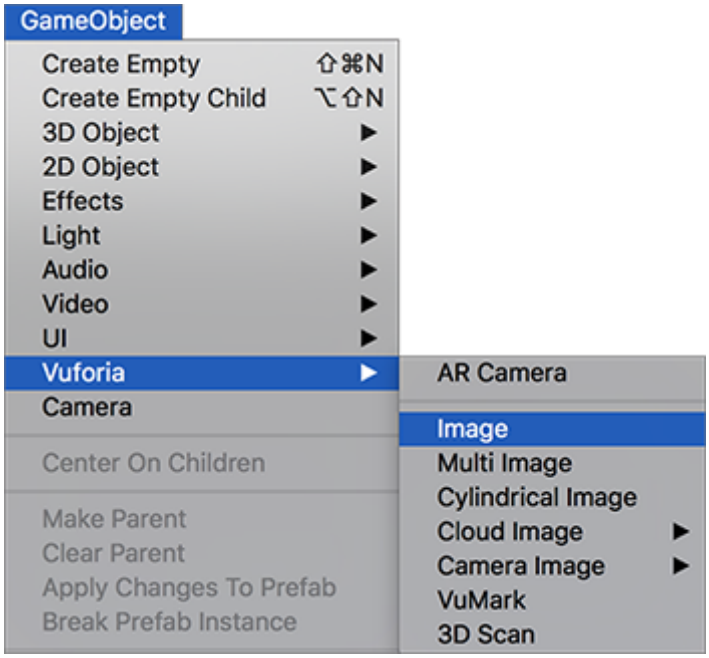
2018–03–28 Page published with editorial review

Vuforia documentation updated for Unity XR API in 2017.3

Adding Image Targets to your Scene

Leave feedback

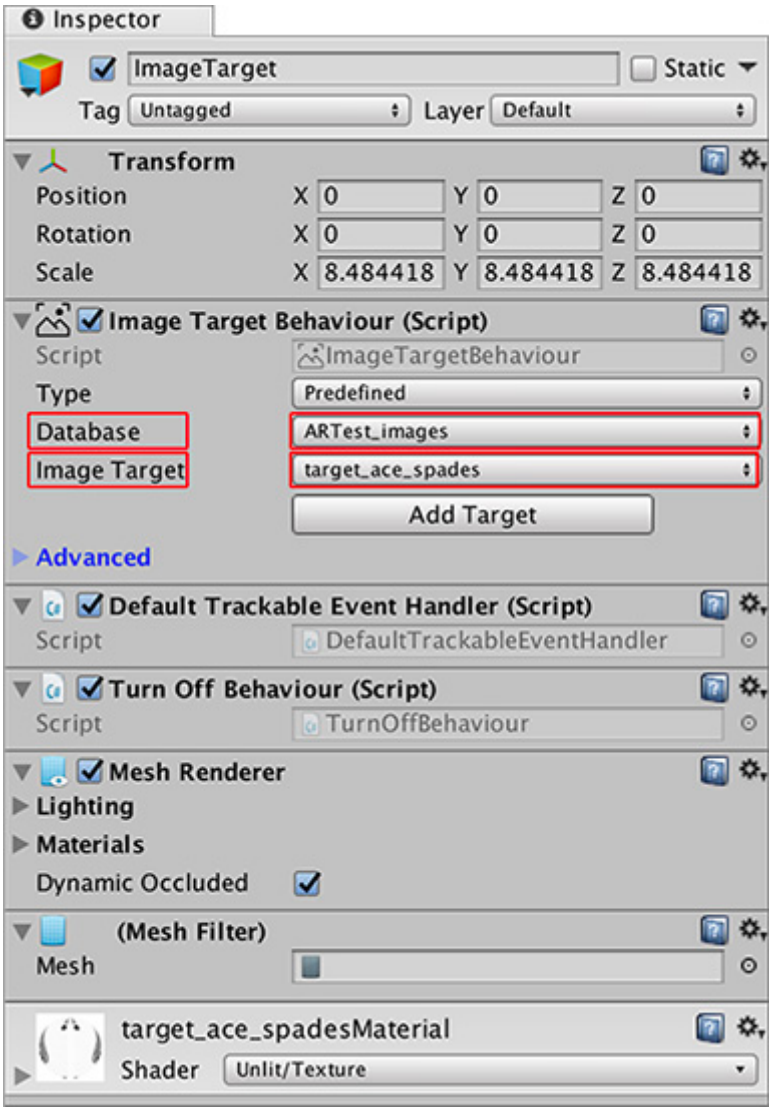
To add an Image Target to the Scene, go to **GameObject** > **Vuforia** > **Image**.



Adding an Image Target GameObject

With the Image Target GameObject in your **Scene**, select it and look at the **Inspector** window to view its **components**.

In the **Image Target Behaviour** component, click on the **Database** drop-down list and select your Target database. In the **Image Target** drop-down list, select the name of your Image Target from the database.



Adding a Target to the Image Target Behaviour component

Note: There is no need to click the **Add Target** button, as this brings you to the Vuforia website to give you information about adding Targets to your app.

The last step is to make a 3D GameObject appear when Vuforia recognizes the Image Target.

2018–03–28 Page published with editorial review

Vuforia documentation updated for Unity XR API in 2017.3

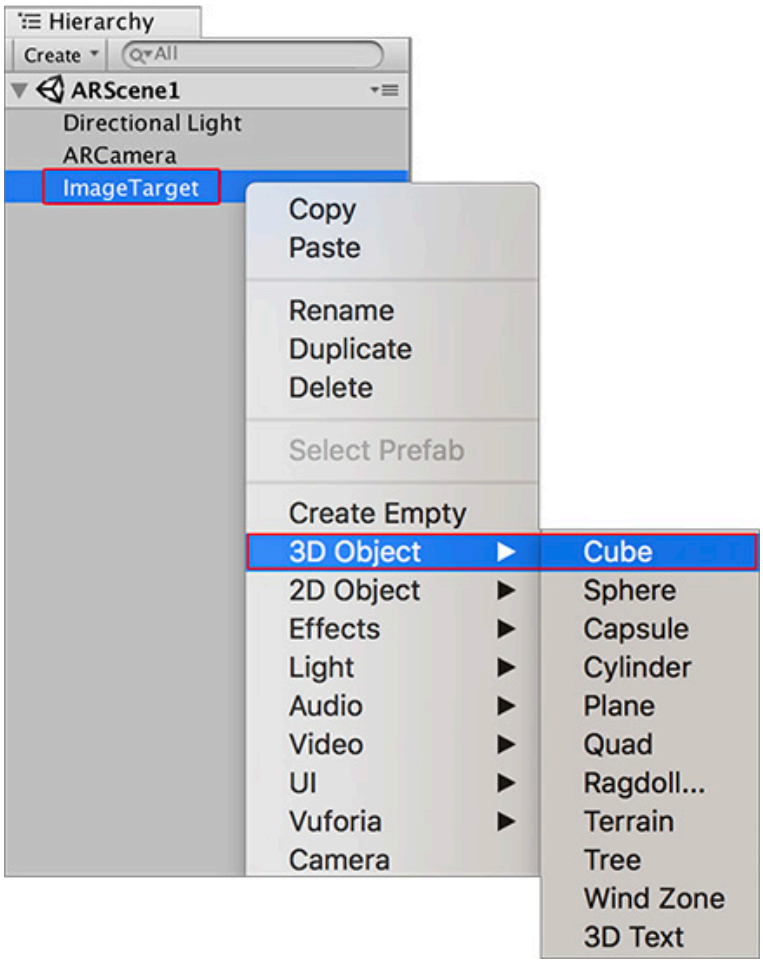
Displaying 3D models on top of tracked images

Leave feedback

This page shows you how to display a **GameObject** when the camera recognizes and tracks an on-screen Image Target.

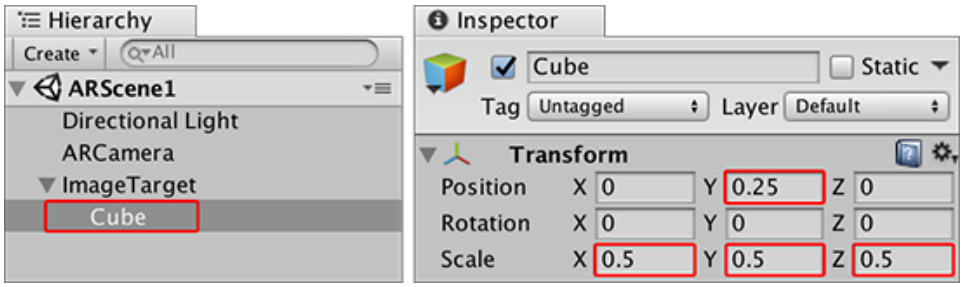
Make the GameObject a child of the Image Target GameObject. This child GameObject must contain both a **MeshRenderer** and **MeshFilter component**.

Add a Cube Primitive as a child to your Image Target GameObject. To do this, right-click it and select **3D Object** > **Cube** from the pop-up menu.



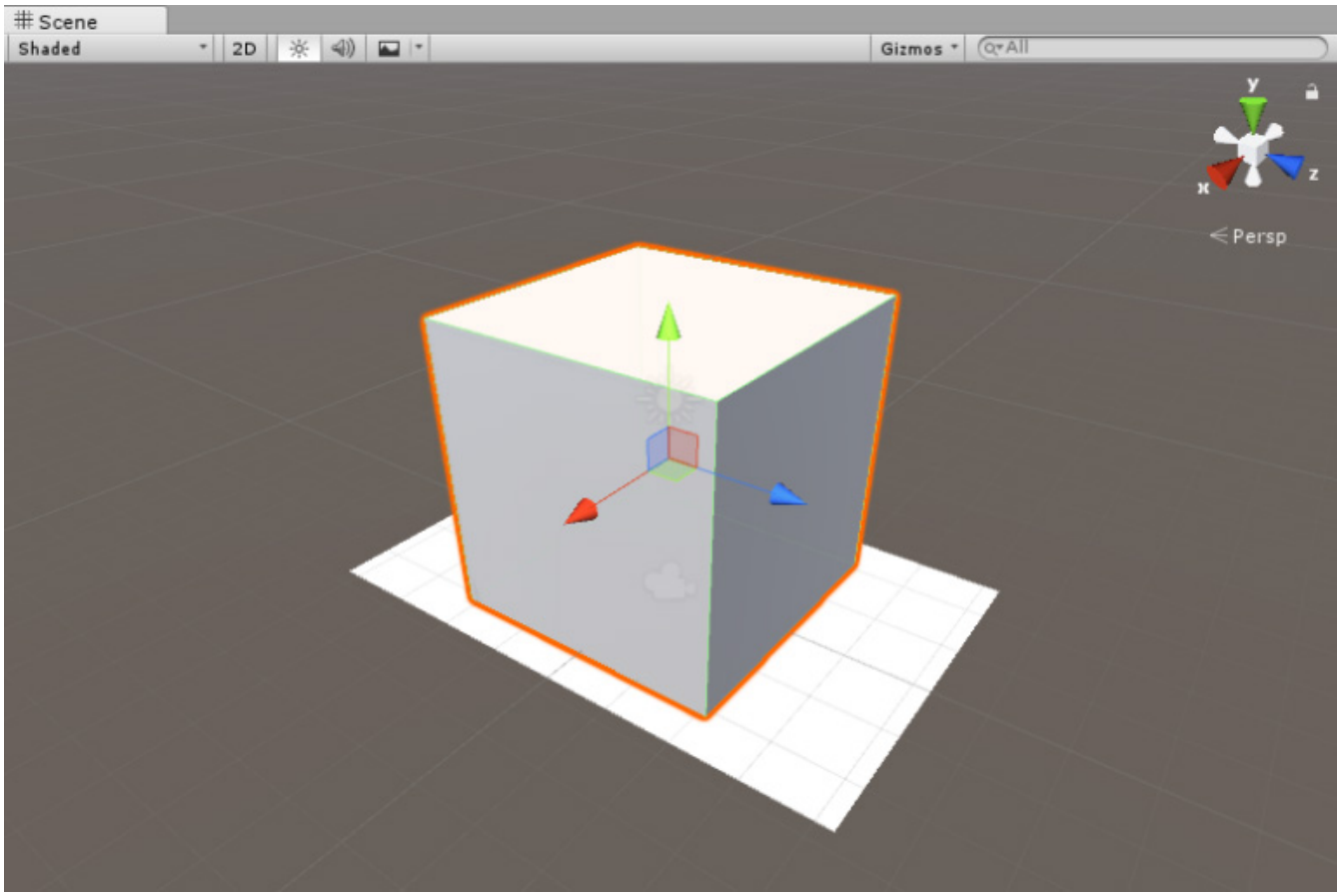
Adding a Cube Primitive as a child to the Image Target GameObject

Now scale the Cube GameObject, and move it closer to your Image Target GameObject to make it look like it is sitting on the Target image. Use the **Scene view** to judge the position of the GameObject. If you used the same **Width** as the playing card in this guide, then change the X, Y and Z Scales of the GameObject's **Transform component** to 0.5 and change the **Transform Position** Y value to 0.25 to make it sit well on the Image Target GameObject.



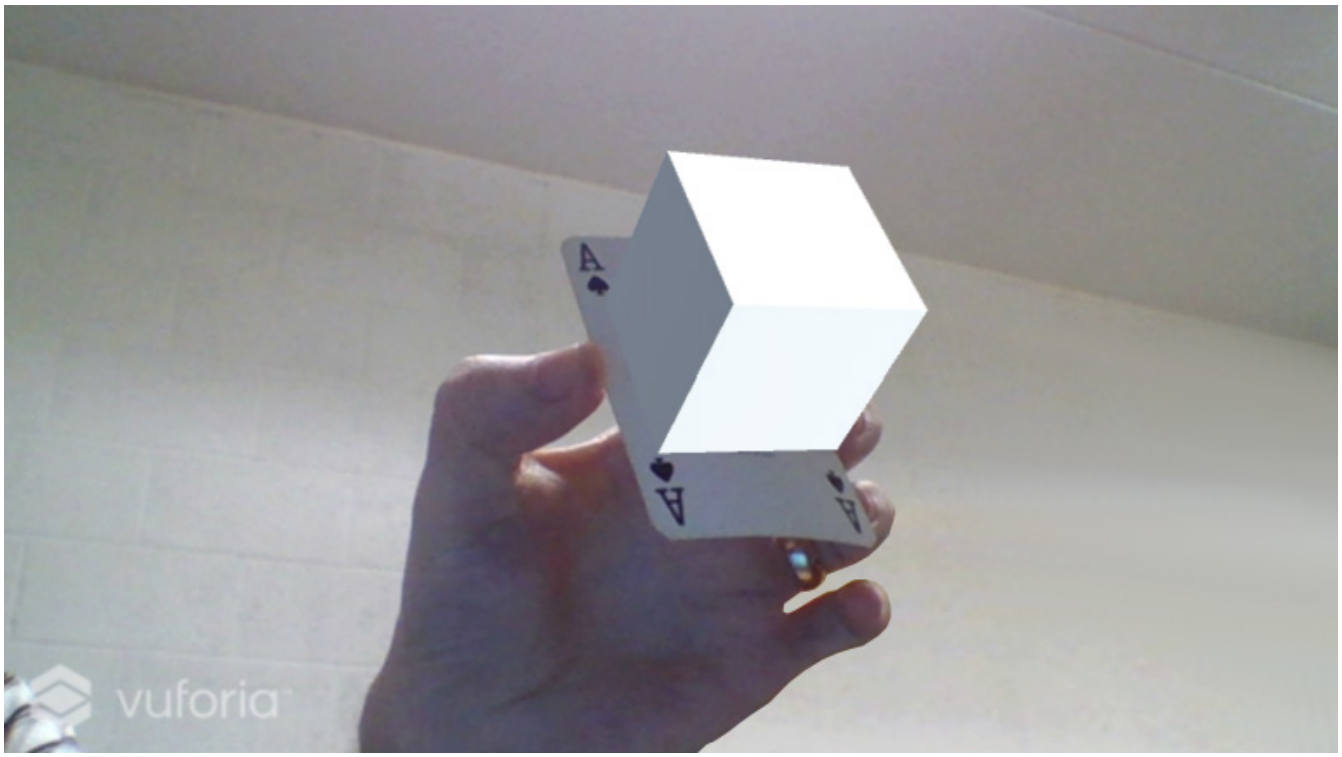
Changing scale and Y position of the cube GameObject

Your **Scene** should look something like this:



Scene view of the Cube on the ImageTarget GameObject

Click the Unity Editor Play button to test the **AR** functionality. When you place the image in front of the webcam, the Cube appears on top of the image in the **Game** view.



Game view showing cube displayed on track image

The Camera component on the **ARCamera** has its default **Far Clipping Planes** set to 2000. For games or applications which require image tracking at larger distances (such as those requiring AR or MR glasses), you need to adjust the **Far Clipping Planes** for the **Camera** component in Unity, and increase the size of the Targets to ensure the device camera can easily track them.

You have now successfully made a simple app that uses an Image Target, and displays a basic 3D shape in its place once the camera can track it.

2018–03–28 Page published with editorial review

Vuforia documentation updated for Unity XR API in 2017.3

Vuforia platform configuration settings

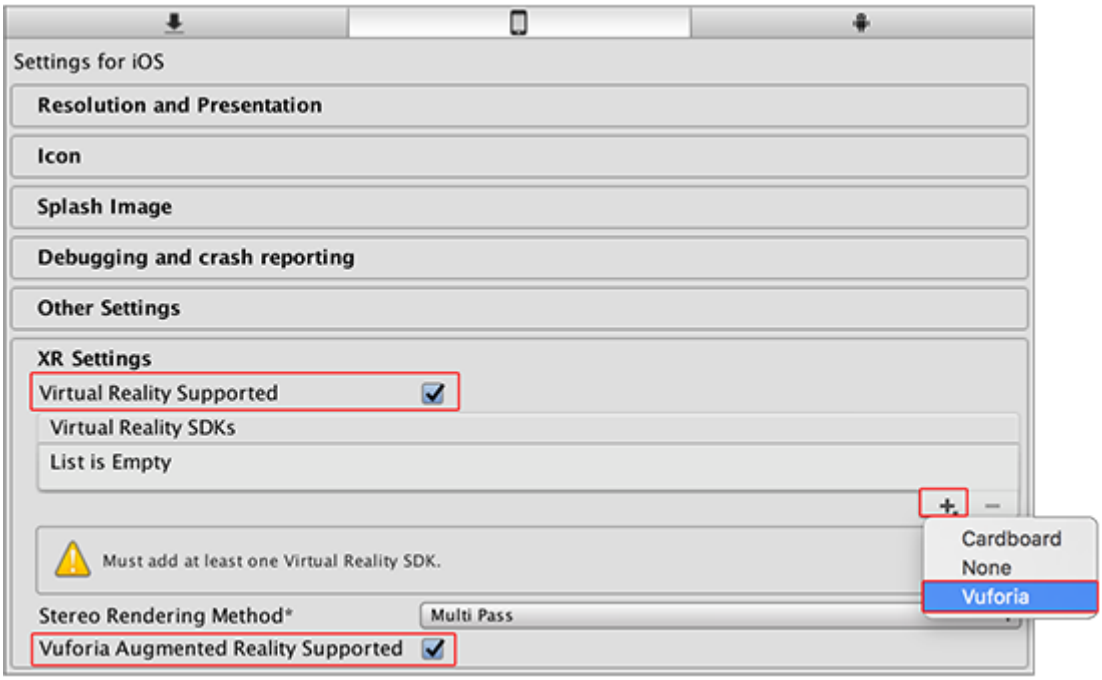
Leave feedback

This section provides information about the important **XR** platform settings affecting development of Vuforia Augmented/Mixed Reality applications with Unity.

Here is a list of important XR settings used in Vuforia's Unity integration. To access these, open the Player Settings (**Edit** > **Project Settings** > **Player**, then select the tab for the device you are building to):

Vuforia Augmented Reality support: Enables Vuforia Augmented Reality support in your application

Virtual Reality SDKs: Allows Vuforia features to be integrated into VR applications. This menu is displayed when you select the **Virtual Reality** Supported checkbox.



Vuforia Augmented Reality Support and Virtual Reality SDKs list in XR Settings

Vuforia supports the following XR SDKs:

Google Cardboard through Google VR SDK

Google Daydream through Google VR SDK

Microsoft HoloLens through Windows 10 SDK

Vuforia SDK

The **Vuforia VR SDK** (in the **Virtual Reality SDKs** list) is a stand-alone VR configuration with no external dependencies. It provides stereo **rendering** and distortion correction, along with head and hand tracking (with positional tracking available for Tango), and viewer profile support to define parameters of various VR headwear.

Stereo rendering method

The stereo rendering method is only relevant when using the Vuforia Virtual Reality SDK. Android, iOS, and **Windows Mixed Reality** devices support Multi-Pass instancing, Single-Pass Instancing and non-instancing.

When using Vuforia Augmented Reality with another Virtual Reality SDK, stereo rendering support is determined by the Virtual Reality SDK being used.

Supported graphics APIs

Recommendation: In Unity's Player Settings (**Edit** > **Project Settings** > **Player**, then select the tab for the device you are building to), open **Other Settings** and enable **Use Auto Graphics API**.

Android	iOS	Windows
OpenGL ES 2.0 OpenGL ES 3.x	OpenGL ES 2.0 OpenGL ES 3.x Metal (iOS 8+)	DirectX 11 on Windows 10

Supported features

Vuforia provides a number of features to allow development of **AR**/MR applications.

The table below lists these features along with the OS and device types supporting them.

Feature	Description	OS	Handhelds	Eyewear
Image Targets	Tracks planar images	Android, iOS, UPW	Yes	Yes
Multi Targets	Tracks geometric arrangements of images	Android, iOS, UPW	Yes	Yes
Cylinder Targets	Tracks images wrapped on cylinders and cones	Android, iOS, UPW	Yes	Yes
User Defined Targets	Tracks images captured by users at runtime	Android, iOS, UPW	Yes	Not recommended
Cloud Recognition	Cloud-based Image Targets	Android, iOS, UPW	Yes	Yes
Device Tracking	3 DoF & 6 DoF positional tracking*	Android, iOS, UPW	Yes (Limited to 6 DoF)	Yes (Limited to 6 DoF)
VuMark	Customizable encodable AR markers	Android, iOS, UPW	Yes	Yes
Object Reco	Tracks **3D objects** from scans	Android, iOS, UPW	Yes	Yes
Model Targets**	Track 3D objects from 3D models	Android, iOS, UPW	Yes	Yes
Smart **Terrain**	Track surfaces and geometry in the user's environment	Android, iOS, UPW	Yes	Limited
AR + VR	AR feature support for VR apps	Android, iOS, UPW	Yes	Yes

***** 6 degrees of freedom tracking is available for Tango and **HoloLens** devices

****** Model Targets are available through the Vuforia Early Access Program and public in Unity 2017.3

Vuforia tips, troubleshooting and resources

Leave feedback

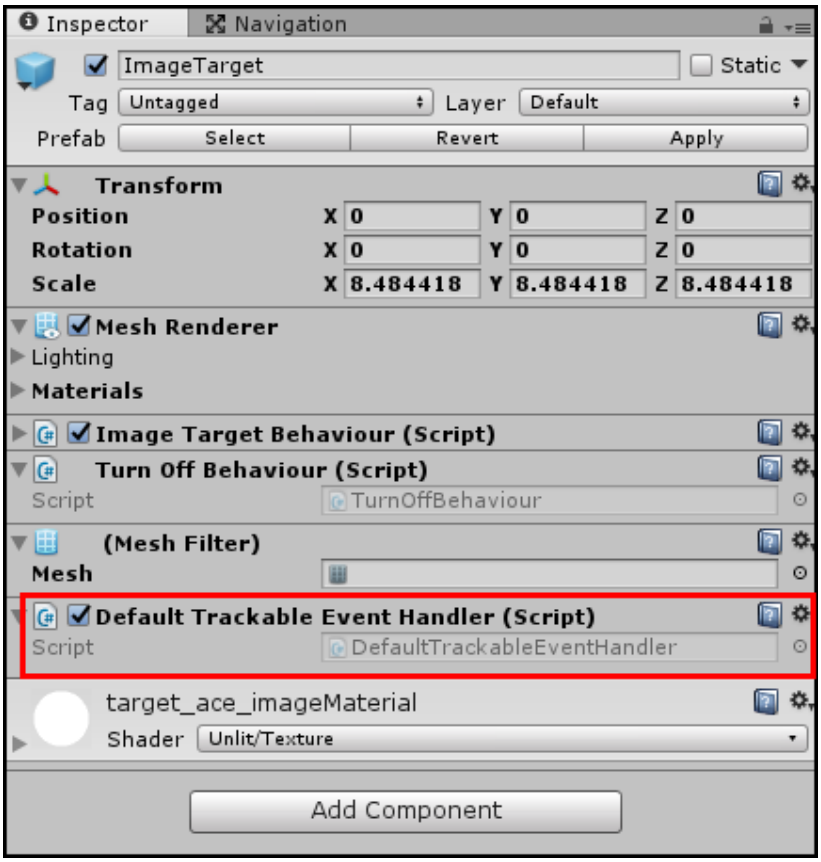
This page provides useful tips to help you make **AR**/MR applications with Vuforia in Unity.

Image Target tracking

When a **camera** is tracking an Image Target in the in-Editor Play mode, Unity disables all **components** belonging to the Image Target GameObject's child **GameObjects**. This does not include any Script **components** attached to the child **GameObjects** of the Image Target GameObject. Any **Scripts** continue to run even when the Image Target is not in view. This might require you to do conditional checks to prevent any code in the Script's `Update()` method constantly running if you don't need it. Alternatively, you can disable the Script component in code, and re-enable it whenever it is needed again.

Running code during Image Target state change events

A useful script for running code during specific Image Target tracking event states (such as whether the target is visible or not) is the **Default Trackable Event Handler (Script)** component attached to each Image Target GameObject.



Locating the Default Trackable Event Handler Script

Here are two of the most useful methods:

```
private void OnTrackingFound()
```

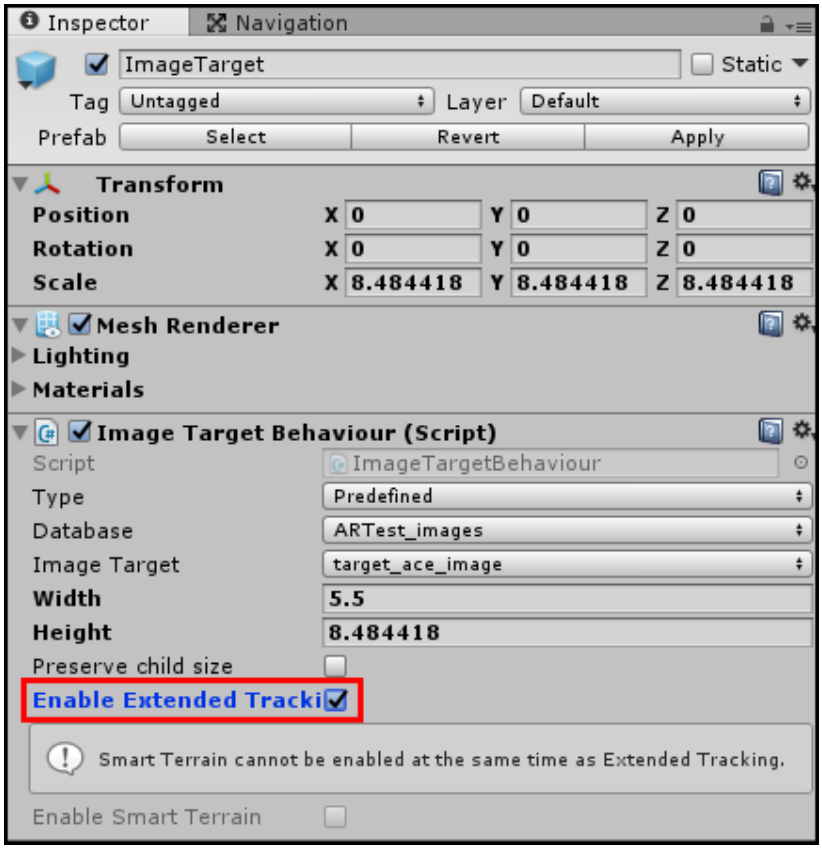
Unity calls this method from the **Default Trackable Event Handler (Script)** component of the specific instance of an Image Target GameObject when Vuforia finds it in the **Camera view**. This method is very useful for running specific code at the very beginning of tracking an object (such as adding the GameObject to a list of active GameObjects).

```
private void OnTrackingLost()
```

Unity calls this method from the **Default Trackable Event Handler (Script)** component of the specific instance of an Image Target GameObject when Vuforia loses track of an Image Target from the Camera view. This method is very useful for running specific code as soon as an Image Target disappears from view (such as removing the GameObject from a list in your GameManager keeping track of all GameObjects active in your application).

Extended Tracking

For Image Targets that only require an initial setup and registration, and do not require images to be constantly tracked, navigate to the Target's **Image Target Behaviour (Script)** component and enable the **Enable Extended Tracking** option.



Enabling Extended Tracking for ImageTarget GameObject

Enable Extended Tracking allows positions and orientations of Image Targets to persist even when not in view of the Camera (after the camera has recognized them at least once) and uses features of the environment to improve tracking performance. More detailed information on Extended Tracking with Vuforia is available in the Vuforia documentation on extended tracking.

Publishing your AR/MR application

To export your Vuforia AR or MR application from Unity to mobile platforms, use the same steps as when normally publishing to Android or **iOS** devices. See documentation on publishing for these platforms:

Publishing to Android
Publishing to iOS
No special settings are required.

Useful links

Here are some useful resources and tutorials to help you learn more about the many features available with Vuforia.

Unity Vuforia forums

Vuforia Developer Forums

Vuforia documentation: Vuforia Developer Library

Vuforia documentation: Best practices for mixed reality AR/VR experiences

Troubleshooting guides

This section provides links to useful troubleshooting information for the most common problems you may encounter when developing with the Vuforia SDK.

[Vuforia Developer Portal: FAQ](https://developer.vuforia.com/forum/faq/faq

---

2018–03–28 Page published with editorial review

Vuforia documentation updated for Unity XR API in 2017.3

Windows Mixed Reality

Leave feedback

Windows Mixed Reality (WMR) is a Microsoft platform built around the Windows 10 API, allowing applications to render digital content on **holographic** and immersive display devices.

Holographic and immersive headsets

Holographic devices such as the Microsoft **HoloLens** allow you to see the physical environment around you while wearing the headset, blending the real world with virtual content.

Windows Mixed Reality immersive headsets feature an opaque display to block out the physical world and surround you in a 360 degree virtual environment.

HoloLens

The HoloLens is a device for running both Augmented Reality and Mixed Reality applications. There are a number of important differences between HoloLens and Windows immersive devices, which developers should consider when creating applications. While HoloLens applications use the same tools and APIs as other **Windows Mixed Reality** applications, their strength lies in allowing you to interact with the real world environment in unique ways.

What are holograms?

HoloLens renders 3D virtual objects in your device's view, creating the illusion that those objects are in the surrounding real-world environment. The user can interact with these objects using gestures, voice commands and gaze. The objects can also interact with real-world surfaces in your physical environment. You can attach audio, animations and other **components** to holograms, as with any other **GameObject** in Unity.

More information on holograms is available on the Microsoft developer website.

Immersive headsets

Windows Mixed Reality (WMR) immersive headsets feature an opaque display to block out the physical world and surround you in a 360 degree virtual environment.

Many **Virtual Reality** devices, such as the HTC Vive or **Oculus** Rift, use sensors built into headsets and **cameras** facing towards these to track user movements. This is known as outside-in tracking. The majority of WMR immersive headsets use inside-out tracking: Instead of using external **cameras** in the environment, inside-out tracking devices use outward-facing **cameras** built into the headset for positional tracking. **Windows Holographic** devices such as the HoloLens also use inside-out tracking.

Differences between Holographic and immersive devices

The table below presents the main features of Holographic and immersive Windows Mixed Reality devices.

Feature	Holographic device	Immersive device
Display	See-through display that allows you to see the physical environment while wearing the headset	Opaque display that blocks out the physical environment while wearing the headset
Tracking Scale	World scale	Stationary or Room scale
Tracking type	Inside-out	Inside-out
Headsets		

The table below lists the important differences between Hololens and immersive devices.

Difference	Details
Holograms do not replace reality	Unlike an immersive 3D environment, holograms are additive; the device draws them over the real world. HoloLens applications augment and can interact with your physical environment.
Input paradigms are different	The primary forms of input for HoloLens are: - **Gaze** (where the user is looking) - **Gesture** (hand signals that the device interprets as commands) - **Voice** (spoken commands). This is a significant shift from conventional input methods.
The HoloLens device is power sensitive	HoloLens is a mobile device, so applications need to be responsible with power usage, just like a phone app. Ensuring you disable features and optimize for CPU utilization is more important than on other platforms.
Photo capture and video capture	Only HoloLens allows the user to capture pictures and videos from the device's camera. Immersive Windows Mixed Reality headsets don't allow the user to access the cameras on the device to capture photos or videos. Using the photo and video capture APIs will allow you to capture from a webcam connected to your PC.
Spatial Mapping	HoloLens supports spatial mapping components and APIs. Windows Mixed Reality immersive headsets do not support these.
Tracking Space	Immersive devices work best for standing-scale (stationary) or room-scale experiences, while the HoloLens works at world scale. Users are not limited to a single location, and can walk far beyond their initial location. Applications need to deal with dynamic reference frames and world-locked content.

For information about input control mapping differences, see Windows Mixed Reality Input.

You can also find additional details in Microsoft's Mixed Reality documentation.

2018–03–27 Page published with editorial review

New content added for XR API changes in 2017.3

WMR hardware and software requirements

Leave feedback

This section discusses hardware compatibilities and software requirements for using **Windows Mixed Reality** devices and applications.

Checking compatibility

You can use Microsoft's Mixed Reality PC Check app to see if your PC is ready to run Windows Mixed Reality applications. The check tests your system and reports back on whether or not your PC meets the requirements.

After the check is complete, the results window displays one of the following result messages:

Compatibility check result	Description
You're good to go	Your PC matches the requirements needed to run Windows Mixed Reality
You're nearly there	Some functions might be limited, but Windows Mixed Reality applications can run
Can't Run Mixed Reality	Your PC doesn't meet the requirements needed to run Windows Mixed Reality

The results window also displays a green tick beside any of your PC components or drivers that meet Microsoft's minimum requirements. It also displays a red 'X' beside the components on your PC that do not meet minimum requirements. The following image below illustrates this:

Windows Mixed Reality compatibility check results screen. In this example, the graphics driver and the RAM are insufficient for Windows Mixed Reality.

The table below lists the minimum and recommended hardware and software requirements to successfully run Windows Mixed Reality applications on a Windows desktop PC. The recommended requirements let you run Windows Mixed Reality

applications on a Windows desktop PC at the highest quality settings.

Component	Minimum requirements	Recommended requirements
Operating System	Windows 10 Fall Creators Update (RS3) - Home, Pro, Business, Education.	Windows 10 Fall Creators Update (RS3) - Home, Pro, Business, Education.
CPU Processor	Intel Core i5 7200U (7th generation mobile), dual-core with Intel® Hyper-Threading Technology enabled (or better).	Intel Core i5 4590 (4th generation), quad-core (or better) AMD Ryzen 5 1400 3.4Ghz (desktop), quad-core (or better).
Disk Space	At least 10 GB.	At least 10 GB.
RAM	8GB DDR3 dual channel (or better).	8GB DDR3 dual channel (or better).
Bluetooth (for controllers)	Bluetooth 4.0	Bluetooth 4.0
USB connectivity	USB 3.0 Type-A or Type-C	USB 3.0 Type-A or Type-C
Graphics Driver	Windows Display Driver Model (WDDM) 2.2	Windows Display Driver Model (WDDM) 2.2
Graphics Card	Integrated Intel® HD Graphics 620 (or greater) DX12-capable integrated GPU (check if your model is greater) NVIDIA MX150/965M (or greater) DX12-capable discrete GPU.	NVIDIA GTX 960/1050 (or greater) DX12-capable discrete GPU. AMD RX 460/560 (or greater) DX12-capable discrete GPU. GPU must be hosted in a PCIe 3.0 x4+ Link slot.
Graphics Display Port	HDMI 1.4 or DisplayPort 1.2	HDMI 2.0 or DisplayPort 1.2

For additional details and information relating to Microsoft's Mixed Reality PC requirements, see Microsoft's Mixed Reality hardware compatibility guidelines.

2018–03–27 Page published with editorial review

New content added for XR API changes in 2017.3

WMR quick start guide

Leave feedback

Windows Mixed Reality (WMR) applications are Universal Windows Applications, and follow the same general guidelines as other Windows Store applications. The development environment is Visual Studio 2017, and does not require an additional SDK installation (Visual Studio installs the necessary SDKs).

Installing required tools

You must install a number of tools on your PC and headset before you can develop Windows Mixed Reality applications. These tools include:

Visual Studio 2017 with required workloads installed (Universal Windows Platform development and Game development with Unity).

HoloLens Emulator and Holographic Templates (Only required for HoloLens development).

To ensure that you install all of the required tools for developing Windows Mixed Reality applications (on both HoloLens and immersive headsets) follow the instructions from Microsoft's documentation on Installing the Tools.

For detailed information on developing for Windows Mixed Reality, see Microsoft's WMR documentation.

After you have installed the required tools, follow the short guide below to set up a Unity project and publish on Windows Mixed Reality devices.

Project set up

A Unity project for Windows Mixed Reality is very similar to a Unity project for other platforms, with a few notable exceptions. To fully support Windows Mixed Reality features, you need to change **camera**, performance and publishing settings as outlined in the following sections.

Camera settings

Ensure that the **Tag** for the Camera you intend to use to track the HMD's position is set to **MainCamera**. To check this, select the Camera you want to use, and in the **Inspector** window, see the **Tag** drop-down. Unity automatically applies this tag to the default Camera in the **Scene**.

Camera settings for HoloLens

For HoloLens , set the **Clear Flags** property of the **Main Camera** to **Solid Color** rather than the default **Skybox,** and set the **Background** color to black (R = 0, G = 0, B = 0, A = 0). You should also make sure the Camera's **Transform** position is set to (0, 0, 0).

Performance settings

For HoloLens only, use the **Fastest** quality setting in **Edit** > **Project Settings** > **Quality**. This maximizes performance and reduces power consumption. In particular, avoid soft shadows and shadow cascades, because they require too many

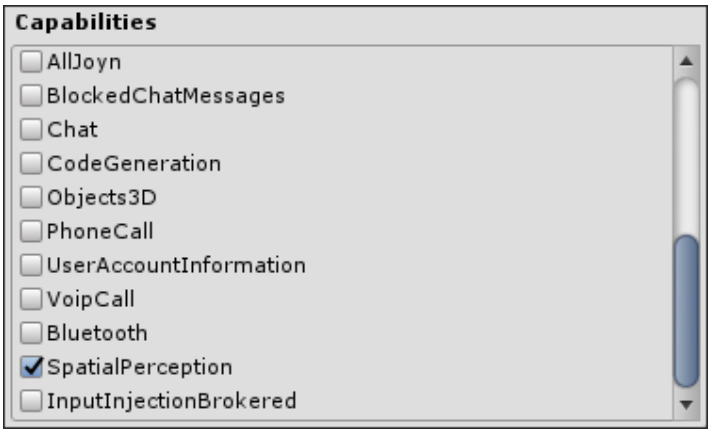
resources to use on the HoloLens.

Setting quality settings for HoloLens

To learn more about optimizing your Windows Mixed Reality application, see Microsoft's documentation on Performance recommendations for Unity.

Publishing settings

To enable important system capabilities for a Windows Mixed Reality application, go to **Player Settings** > **Publishing Settings** and check the box for each option you want to use from the **Capabilities** list.



System capability settings found in **Player Settings** > **Publishing Settings**

Note: Not all of the publishing settings in the **Capabilities** list are specific to Windows Mixed Reality. For more information, see WSA Player Settings.

The following table describes the important capability settings for publishing Windows Mixed Reality applications:

Capability Setting	Immersive headset support	HoloLens support	Description
InternetClient	Y	Y	Provides access to your Internet connection for outgoing connections to the Internet. WMR requires this for voice recognition.
InternetClientServer	Y	Y	Provides access to your Internet connection, including incoming unsolicited connections from the Internet. The app can send information to or from your computer through a firewall. WMR requires this for voice recognition. Note: If this is active, you don't need to use **InternetClient**.
MusicLibrary	Y	N	Provides access to your music library and playlists, including the capability to add, change, or delete files. This allows you to use **VideoCapture** audio recording functionality in your WMR app.
PicturesLibrary	Y	Y	Your picture library, including the capability to add, change, or delete files. This capability also includes pictures libraries on HomeGroup computers, along with picture file types on locally connected media servers.

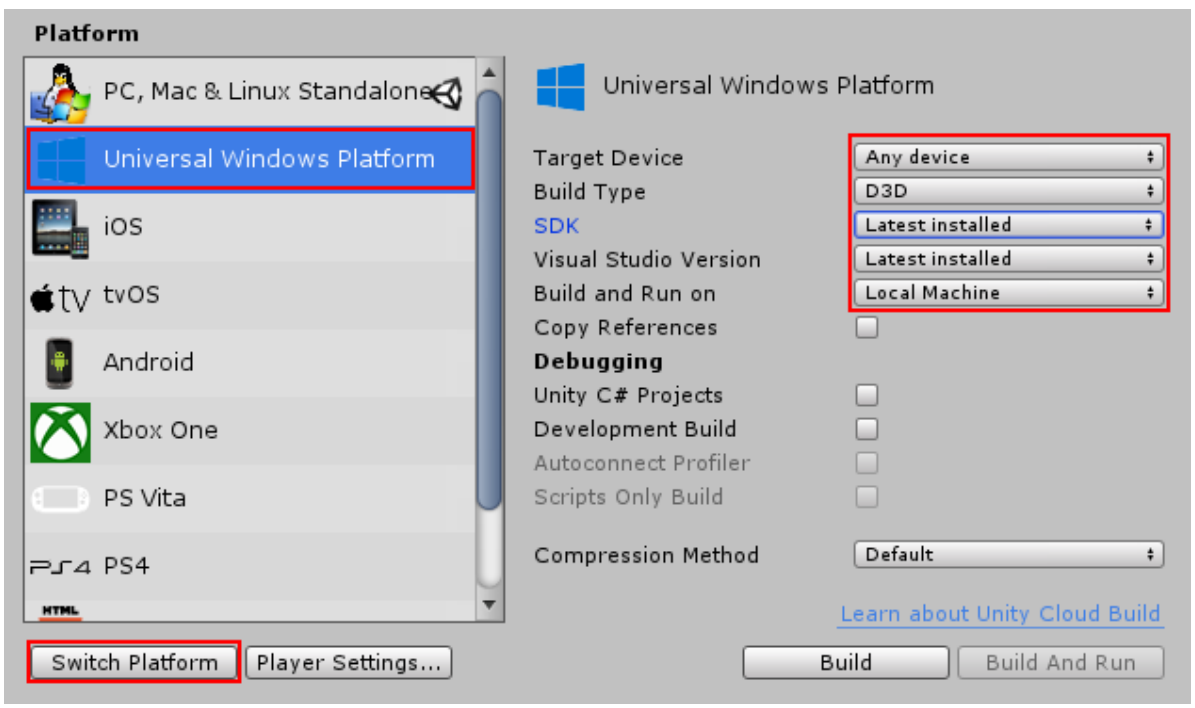
Capability Setting	Immersive headset support	HoloLens support	Description
VideosLibrary	Y	N	Your video library, including the capability to add, change, or delete files. This capability also includes video libraries on HomeGroup computers, along with video file types on locally connected media servers.
WebCam	N	Y	Allows you to use **PhotoCapture** and **VideoCapture** functionality in your app.
Microphone	Y	Y	Allows you to use voice recognition functionality in your app.
Bluetooth	Y	Y	Enables bluetooth communication in your app. WMR requires this to allow you to use Windows Mixed Reality spatial controllers.
SpatialPerception	N	Y	Allows you to use Spatial Mapping in your app.

For more detail on these capabilities, see the Microsoft documentation.

Exporting a Visual Studio solution

When you have created your project and are ready to test it, export the project to a Visual Studio solution. To deploy your WMR application you must first build a Visual Studio solution with Unity.

Go to **File** > **Build Settings** and select **Universal Windows Platform** from the **Platform** list. Now click the **Switch Platform** button at the bottom left of the window to configure the Editor to build for Windows.



Build Settings window showing Universal Windows Platform default settings

For a standard build, the default settings work correctly for Windows Mixed Reality immersive headsets.

To build for HoloLens, you should change the **Target Device** setting to **HoloLens**.

The table below provides a list of the Build Settings available for the Universal Windows Platform and describes their usage.

Setting	*Description*
Target Device	Choose **Any Device** to build for immersive headsets and **HoloLens** to build for HoloLens. This is important for optimisation.
Build Type	Choose from D3D (Direct3D) or XAML.
D3D	Gives faster results than XAML because there is no XAML layer in the app. This builds the app in 3D exclusive space, and you can't switch to a 2D XAML app or modify this after generation.
XAML	Adds an XAML code layer over the app, which allows the user to switch from the 3D app and open a 2D app. XAML code can be modified after generation. The most common example of this is using a touch keyboard for the HoloLens.
SDK	Select the specific version of Windows 10 SDK your app uses. This is set to **Latest installed** by default.
Visual Studio Version	Select the specific version of Visual Studio to generate a solution (.sln) for. This is set to **Latest installed** by default, which is the recommended setting if you have Visual Studio 2017 installed.
Build and Run on	Choose which device your application runs on when you click the **Build and Run** button. This is set to **Local Machine** by default, and you should not need to change this. **Note**: Unity ignores this option when you click the **Build** button.
Copy References	Enables copying of UnityPlayer.dll, associated dlls and data to the generated solution folder instead of referencing them directly from Unity installation folder. This requires extra space, but enables the generated solution to be portable (you can copy it to another machine and build it even if Unity is not installed on that machine).
Unity C# Projects	Allows inclusion of the scripting files from your project in the generated solution. This setting is only available when you set the **Scripting Backend** to **.NET**.
Development Build	Unlocks debug features and allows developers to test and debug built projects. This allows you to connect the built project to the Unity **profiler**, and provides other development features such as more verbose debug log outputs. Enabling this setting also makes the **Scripts Only Build** and **Autoconnect Profiler** settings available.
Autoconnect Profiler	Allow Unity's built-in profiler to automatically connect to the build. This is only available when **Development Build** is enabled.
Scripts Only Build	Build only the **scripts** of a project. This is only available when **Development Build** is enabled.

Before you can build your app, you should configure your Player Settings to build Windows Mixed Reality applications correctly:

From the Build Settings window, click **Player Settings** > **XR Settings**.

Enable **Virtual Reality Supported**.

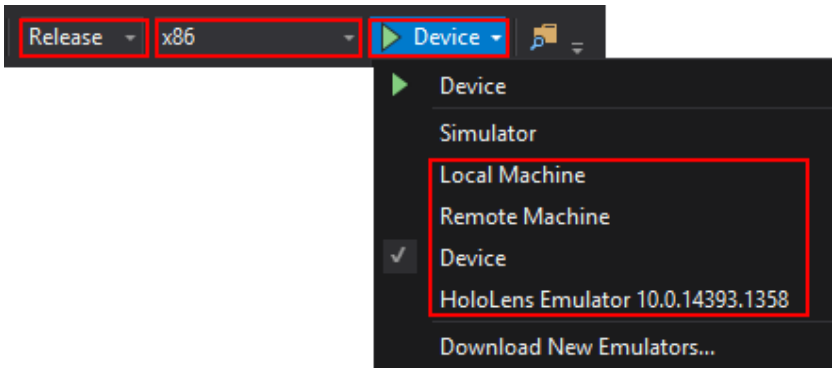
Click the **+** button on the **Virtual Reality Devices** list and select **Windows Mixed Reality**.

After you have configured your Build Settings, click the **Build** button. Build your project to a new folder and remember the location.

To deploy your app from Visual Studio, do the following:

In Visual Studio, open the generated solution file (.sln) located inside the folder where you built your project.

In the main Visual Studio taskbar (left to right in image below), change the target platform for your solution, and select which device to run the solution on.



Click the dropdown arrow on the right of the Run button (marked by a green arrowhead) and a list of possible devices will appear.

There are four main options to test and run your Windows Mixed Reality application from Visual Studio:

Local Machine
Remote Machine
Device
HoloLens Emulator
The sections below describe these options in more detail. Not all of these options work for all WMR devices; some are specific to HoloLens or immersive headsets.

Local Machine (immersive headsets only)

Local Machine allows you to build your application and install it to the Mixed Reality Portal on your Windows 10 PC. When built, your app automatically runs on your PC, and you can test it through your immersive headset.

You can launch your application again at any time through the Start menu from the Mixed Reality Portal.

Remote Machine (HoloLens only)

Remote Machine prompts you to enter the IP address of the HoloLens or other headset you wish deploy to. When you click **Run** with **Remote Machine** selected, a dialog box asks you for a PIN for the device.

To get this PIN:

Switch on your HoloLens and go to **HoloLens Settings**.

Select the **For Developers** tab and enable **Developer Mode** using the provided toggle button.

Select the **Pair** button to get the **PIN** and enter it into Visual Studio in the popup box.

Your application remote installs onto your HoloLens and, when the build process is complete, runs on the device automatically.

Device (HoloLens only)

Device allows you to build your Visual Studio project and deploy it to a HoloLens device connected to your PC via a USB cable.

When you click **Run**, Visual Studio builds the project, installs the app on the connected HoloLens, and runs on the device automatically once the build process is complete.

HoloLens Emulator (HoloLens only)

HoloLens Emulator allows you to build the Visual Studio project and run the app on an installed HoloLens emulator. This emulator lets you test your app and simulate gestures and other inputs on your PC before you deploy it to a HoloLens device.

Including Scripts in your generated solution (optional)

Tick the **Unity C# Projects** checkbox to include the scripting files from your project in the generated solution. This allows you to edit and debug your scripts without needing to re-export from Unity. You only need to re-export when your project settings or content change. This setting is only available when you set the **Scripting Backend** to **.NET**.

You can use either **IL2CPP** or **.NET** scripting backends for your application. To change the **Scripting Backend**, go to **Player Settings** > **Other Settings** and select the relevant backend in the **Configuration** section.

For more information on ILCPP, see Unity's documentation on How IL2CCP works.

For more information on building to Windows Mixed Reality devices, see Microsoft's documentation on Exporting and building a Unity Visual Studio solution.

2018–03–27 Page published with editorial review

New content added for XR API changes in 2017.3

WMR input and interaction concepts

Leave feedback

All **Windows Mixed Reality** devices use specific forms of interaction to take input from the user. Some of these forms of input are specific to certain **Windows Mixed Reality** devices, such as **HoloLens** or immersive headsets.

Immersive headsets use a variety of inputs, including spatial controllers. HoloLens is limited to using 3 forms of input, unless extra hardware is required by the application to be used in addition to the headset.

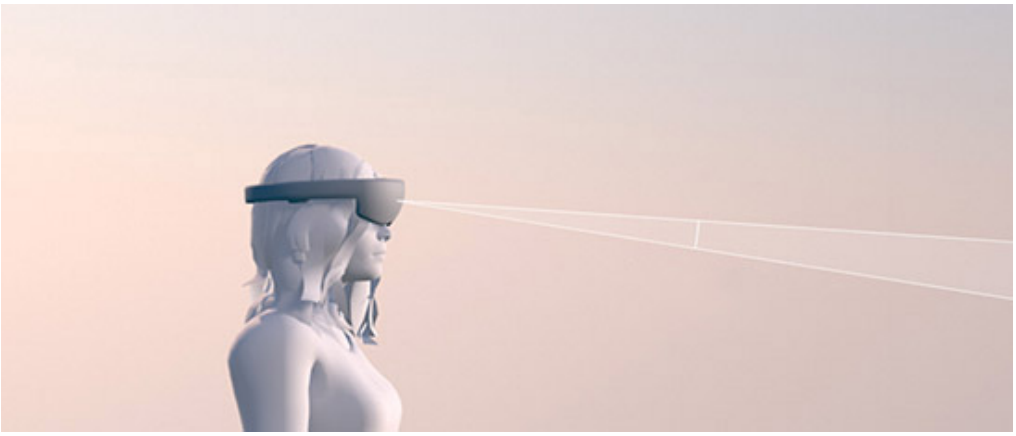
All forms of input work with Windows Mixed Reality immersive headsets. Gaze works in VR, gestures trigger when you use the Select button on the controller, and voice is available when the end user has a microphone connected to their PC.

Input on HoloLens is different from other platforms because the primary means of interaction are:

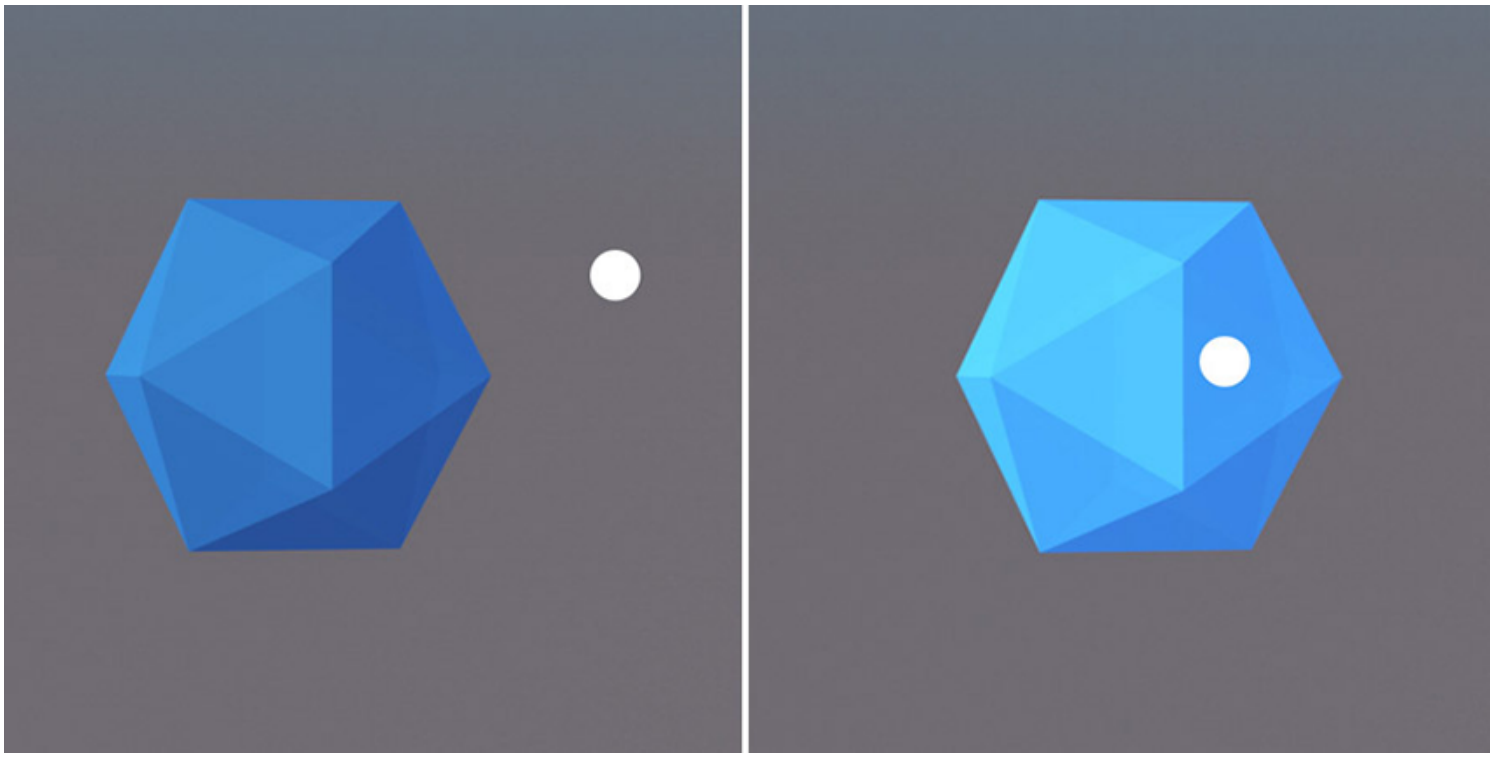
Gaze: Controls the application by tracking where the user is looking in the world.
Voice: Short, spoken commands, and longer free-form dictation, which can be used to issue commands or trigger actions.
Gesture: Hand signals or controller input trigger commands in the system.

Gaze (all devices)

Gaze is an input mechanism that tracks where a user is looking:



On HoloLens, this is accurate enough that you can use it to get users to select **GameObjects** in the world. You can also use it to direct commands at specific **GameObjects** rather than every GameObject in the **Scene**.



Example of targeting a GameObject using Gaze (the white circle represents the Gaze indicator)
For more information, see Microsoft's documentation on Gaze indicator and Gaze targeting.

Voice (all devices)

Windows 10 API provides voice input on both HoloLens and immersive devices. Unity supports three styles of input:

Keywords: Simple commands or phrases (set up in code) used to generate events. This allows you to quickly add voice commands to an application, as long as you do not require localization. The KeywordRecognizer API provides this functionality.

Grammars: A table of commands with semantic meaning. You can configure grammars through an XML grammar file (.grxml), and localize the table if you need to. For more information on this file format, see Microsoft's documentation on Creating Grammar Files. The GrammarRecognizer API provides this functionality.

Dictation: A more free-form text-to-speech system that translates longer spoken input into text. To prolong battery life, dictation recognition on the HoloLens is only active for short periods of time. It requires a working Internet connection. The DictationRecognizer API provdes this functionality.

HoloLens headsets have a built in microphone, allowing voice input without extra hardware. For an application to use voice input with immersive headsets, users need to have an external microphone connected to their PC.

For more information about voice input, refer to Microsoft's documentation on Voice design.

Gestures (all devices)

A gesture is a hand signal interpreted by the system or a controller signal from a Spatial Controller. Both HoloLens and immersive devices support gestures. Immersive devices require a spatial controller to initiate gestures, while HoloLens gestures require hand movement. You can use gestures to trigger specific commands in your application.

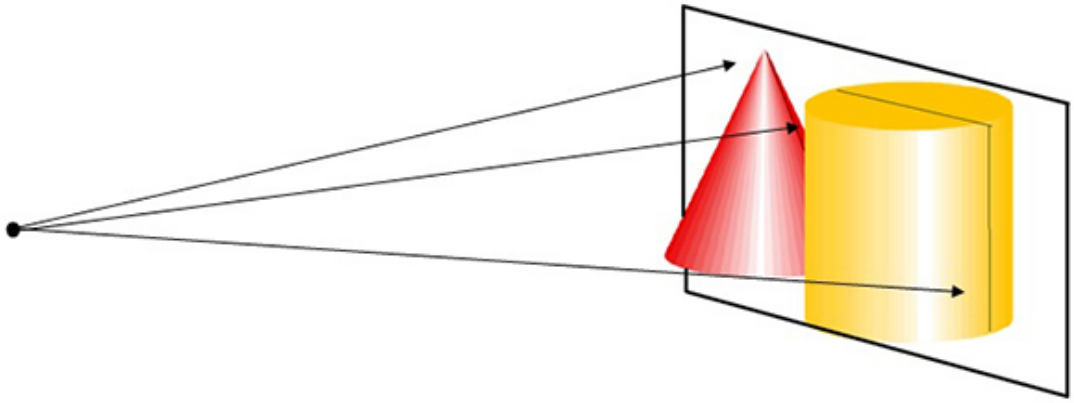
Windows Mixed Reality provides several built-in gestures for use in your application, as well as a generic API to recognize custom gestures. Both built-in gestures and custom gestures (added via API) are functional in Unity.

For more information, see Microsoft's documentation on gestures.

Focus Point (all devices)

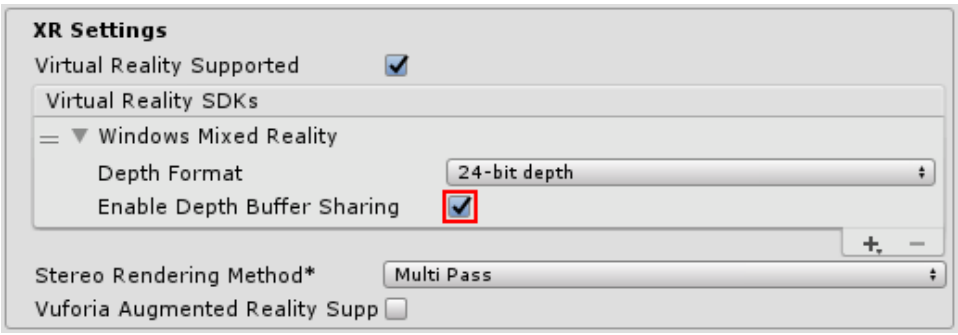
Windows Mixed Reality uses a process called Late Stage Reprojection 9LSR) to compensate for movement of the user's head between the time a frame is rendered and when it appears on the display. To compensate for this latency, LSR modifyies the rendered image according to the most recent head tracking data, then presents the image on the display.

The HoloLens uses a stabilization plane for reprojection (see Microsoft documentation on plane-based reprojection). This is a plane in space which represents an area where the user is likely to be focusing. This stabilization plane has default values, but applications can also use the SetFocusPoint API to explicitly set the position, normal, and velocity of this plane. Objects appear most stable at their intersection with this plane. Immersive headsets also support this method, but for these devices there is a more efficient form of reprojection available called per-pixel depth reprojection.



Stabilization plane for 3D objects

Desktop applications using immersive headsets can enable per-pixel depth reprojection, which offers higher quality without requiring explicit work by the application. To allow per-pixel depth reprojection, open the Player Settings for the Windows Mixed Reality, go to **XR Settings > Virtual Reality SDKs** and check **Enable Depth Buffer Sharing**.



Enabling per-pixel reprojection on desktop hardware using immersive headsets

When **Enable Depth Buffer Sharing** is enabled, ensure applications do not explicitly call the `SetFocusPoint` method, which overrides per-pixel depth reprojection with plane-based reprojection.

Enable Depth Buffer Sharing on HoloLens has no benefit unless you have updated the device's OS to Windows 10 Redstone 4 (RS4). If the device is running RS4, then it determines the stabilization plane automatically from the range of values found in the depth buffer. Reprojection still happens using a stabilization plane, but the application does not need to explicitly call `SetFocusPoint`.

For more information on how the HoloLens achieves stable holograms, see Microsoft's documentation on hologram stability.

Anchors (HoloLens only)

Anchor components are a way for the virtual world to interact with the real world. An anchor is a special **component** that overrides the position and orientation of the Transform **component** of the GameObject it is attached to.

WorldAnchors

A WorldAnchor represents a link between the device's understanding of an exact point in the physical world and the GameObject containing the WorldAnchor component. Once added, a GameObject with a WorldAnchor component remains locked in place to a location in the real world, but may shift in Unity coordinates from frame to frame as the device's understanding of the anchor's position becomes more precise.

Some common uses for WorldAnchors include:

Locking a **holographic** game board to the top of a table.

Locking a video window to a wall.

Use WorldAnchors whenever there is a GameObject or group of GameObjects that you want to fix to a physical location.

WorldAnchors override their parent GameObject's **Transform component**, so any direct manipulations of the **Transform component** are lost. Similarly, GameObjects with WorldAnchor components should not contain **Rigidbody** components with dynamic physics. These components can be more resource-intensive and game performance will decrease the further apart the WorldAnchors are.

Note: Only use a small number of WorldAnchors to minimise performance issues. For example, a game board surface placed upon a table only requires a single anchor for the board. In this case, child GameObjects of the board do not require their own WorldAnchor.

For more information and best practices, see Microsoft's documentation on spatial anchors.

2018–03–27 Page published with editorial review

New content added for XR API changes in 2017.3

WMR testing during development

Leave feedback

Unity provides tools that allow you to test applications during the development process. These tools allow you to debug your application as if it was any other Unity application, as well as wirelessly pushing content to your device. Some of these tools only work with specific WMR devices, such as **HoloLens**.

Play in Editor (immersive devices)

When using an immersive **Windows Mixed Reality** device, you can test and iterate on your application directly in the Unity Editor. Make sure the **Windows Mixed Reality** Portal is open and the headset is active, then press the **Play** button in the Editor. Now your application runs within the Editor while also **rendering** to the headset.

Known limitations

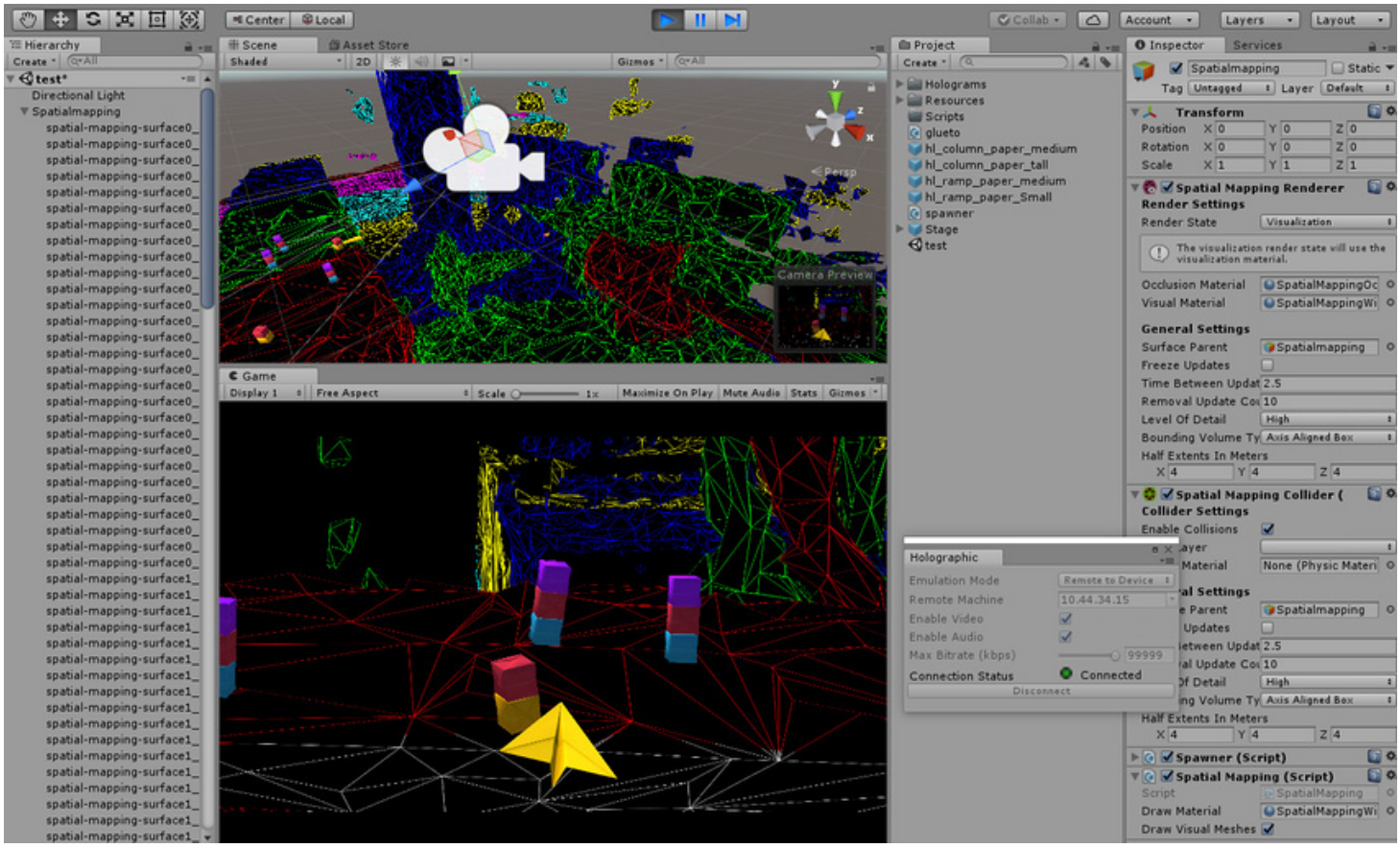
There are a couple of limitations to consider when using Play in the Editor with Mixed Reality devices:

Keyboard, mouse, and spatial controllers work in the Editor, as long as they are recognized by Windows and Unity.

Non-Windows Mixed Reality input devices only work when the game window has focus, otherwise keyboard and mouse controls may not work correctly.

Holographic Emulation (HoloLens only)

Holographic Emulation allows you to prototype, debug, and run Microsoft HoloLens projects directly in the Unity Editor rather than building and running your application every time you wish to see the effect of an applied change. This vastly reduces the time between iterations when developing HoloLens applications in Unity.



The Unity Editor running with Holographic Emulation

Holographic Emulation has three different modes:

Remote to Device: Using a connection to a **Windows Holographic** device, your application behaves as if it were deployed to that device, while in reality it is running in the Unity Editor on your host machine. For more information, see the section Emulation Mode:

Remote to device, below.

Simulate in Editor: Your application runs on a simulated HoloLens device directly in the Unity Editor, with no connection to a real-world Windows Holographic device. For more information, see the section Emulation Mode: Simulate in Editor, below.

None: This setting allows you to still run your application in the Editor if you have a Windows Holographic device.

Holographic emulation is supported on any machine running Windows 10 (with the **Fall Creators** update). Both **Remote to Device** and **Simulate in Editor** aim to mimic the current HoloLens runtime as closely as possible, so newer APIs beyond this are not present. This means that recent additions to input code don't work in these situations, just like they wouldn't on a HoloLens. For example, rotation, position accuracy, angular velocity, and the basis vectors for a pose of forward, up, and right, are all absent, so those APIs fail.

To enable remoting or simulation, open the Unity Editor and go to **Window** > **XR** > **Holographic Emulation**.

This opens the **Holographic** control window, which contains the **Emulation Mode** drop-down menu. You must keep this window visible during development, so that you can access its settings when you launch your application.

Default Holographic Emulation control window

Emulation Mode is set to **None** by default, which means that if you have an immersive headset connection, your application runs in the Editor and plays on the HMD. See the **Play in Editor** section above for more details.

Change the **Emulation Mode** to **Remote to Device** or **Simulate in Editor** to enable emulation for HoloLens development. This overrides Play in Editor support for any connected immersive device.

Emulation Mode: Remote to Device

Holographic Remoting allows your application to behave as if it were deployed to a HoloLens device, while actually running in the Unity Editor on the host machine. Even while remoting, spatial sensor data and head tracking from the connected device are still active and work correctly. The Unity Editor Game view allows you to see what is being rendered on the device, but not what the wearer of the device sees of the real-world. (See an example in the image above titled 'The Unity Editor running with Holographic Emulation'.)

Note: Holographic Remoting is a quick way to iterate over changes during development, but you should avoid using it to validate performance. Your application is running on the host machine rather than the device itself, so you might get inaccurate results.

To enable this mode in the Unity Editor, set the **Emulation Mode** to **Remote to Device**. The Holographic window then changes to reflect the additional settings available with this mode.

Connecting your device

To remote to a HoloLens device, you need to install the Holographic Remoting Player on your HoloLens device.

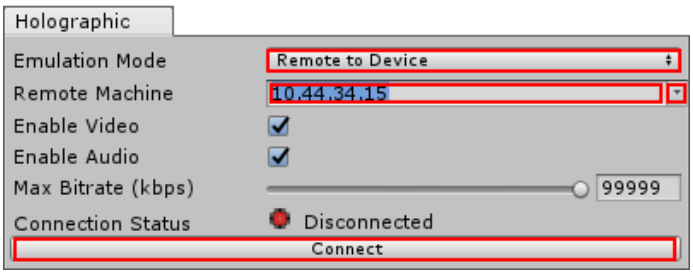
To install the Holographic Remoting Player application and configure the Unity Editor:

Install and run the Holographic Remoting Player from your HoloLens store. This is available from the Windows Store app on your Hololens. When launched, the remoting player enters a waiting state and displays the IP address of the device on the HoloLens screen:

The Holographic Remoting Player screen

For additional information about this Player, including how to enable connection diagnostics, see the Microsoft Windows Dev Center.

Enter the IP address of your HoloLens device into the **Remote Machine** field from the Holographic window in the Unity Editor. The drop-down button to the right of the field allows you to select recently used addresses:



Configuring Remote to Device emulation mode

Click the **Connect** button. The connection status should change to a green light with a connected message.

Click **Play** in the Unity Editor to run your device remotely.

When remoting starts, you can pause, inspect **GameObjects**, and debug in the same way as if you were running any app in the Editor. The HoloLens device and your host machine transmit video, audio, and device input back and forth across the network.

Known limitations

There are a couple of limitations you should consider when using **Remote to Device** emulation with HoloLens:

Unity does not support Speech (PhraseRecognizer) when using **Remote to Device** emulation mode. When in this mode, the emulator intercepts speech from the host machine running the Unity Editor. The application uses the microphone in the PC you are remoting from, instead of the Hololens microphone.

While **Remote to Device** is running, all audio on the host machine redirects to the device, including audio from outside your application.

Emulation Mode: Simulate in Editor

When using this emulation mode, your application runs on a simulated Holographic device directly in the Unity Editor, with no connection to a real-world HoloLens device. This is useful when developing for Windows Holographic if you do not have access to a HoloLens device.

Note: You should test your application on a HoloLens device to ensure it works as expected. Don't rely exclusively on emulation during development.

To enable this mode, set the **Emulation Mode** to **Simulate in Editor** and click the **Play** button. Your application then starts in an emulator built into the Unity Editor.

In the Holographic Emulation control window, use the **Room** drop-down menu to choose an available virtual room (the same as those supplied with the XDE HoloLens Emulator). Use the **Gesture Hand** drop-down menu to specify which virtual hand performs gestures (left or right).

Configuring Simulate in Editor emulation mode

In **Simulate in Editor** mode, you need to use a **game controller** (such as an Xbox 360 or Xbox One controller) to control the virtual human player. The simulation still works if you do not have a controller, but you cannot move the virtual human player around.

The table below lists the controller inputs and their usage during **Simulate in Editor** mode.

Control	Usage
Left-stick	Up and down move the virtual human player backward and forward. Left and right move the human player left and right.
Right-stick	Up and down rotate the virtual human player's head up and down (rotation about the X axis - Pitch). Left and right turn the virtual human player left and right (rotation about the Y axis - Yaw).
D-pad	Move the virtual human player up and down or tilt the player's head left and right (rotation about the Z axis - Roll).
Left and right trigger buttons or A button	Perform a tap gesture with a virtual hand.
Y button	Reset the pitch (X rotation) and roll (Z rotation) of the virtual human player's head.

To use a game controller, focus the Unity Editor on the Game view. If you are currently focused on another Unity Editor window, click on the Game view window to refocus on the Game view.

Known limitations

There are a couple of limitations you should consider when using **Simulate in Editor** emulation with HoloLens:

Most game controllers work in **Simulate in Editor** mode, as long Windows and Unity recognise them. However, unsupported controllers might cause compatibility issues.

You can use PhotoCapture during **Simulate in Editor** mode, but you must use an externally connected **camera** (such as a webcam) due to the lack of a connected HoloLens device. This also prevents you from retrieving a matrix with `TryGetProjectionMatrix` or `TryGetCameraToWorldMatrix`, because a normal external camera cannot compute where it is in relation to the real world.

Useful resources and troubleshooting

When troubleshooting development issues for Windows Mixed Reality applications, you might find these external resources useful:

Windows Mixed Reality development official documentation

Microsoft Windows Mixed Reality Tutorials

Microsoft Windows Mixed Reality immersive headset development tutorial

Microsoft HoloLens frequently asked questions (FAQ)

Microsoft HoloLens known issues

Microsoft Windows Mixed Reality troubleshooting guides

Unity Forum post on WMR Tech Preview

2018–03–27 Page published with editorial review

New content added for XR API changes in 2017.3

HoloLens WorldAnchor persistence

Leave feedback

Persistence is a system for saving World Anchor states across multiple runs of the same application. This can be thought of as a “save game” functionality for physical locations in the real world. An example of this is remembering where a game board is placed in the world when the application re-launches.

The `WorldAnchorStore` provides basic functionality for saving and loading World Anchors. Retrieve a `WorldAnchorStore` by calling `WorldAnchorStore.GetAsync` and providing it with a callback. Your callback saves the `WorldAnchorStore` returned so that it can be used for future operations.

To save an existing World Anchor, give it a name and call the `Save` function on the `WorldAnchorStore`. See an example below:

```
private void SaveAnchor()
{
    if (!this.savedAnchor) // only save this once
    {
        this.savedAnchor = this.MyWorldAnchorStore.Save("MyAnchor", MyWorldAncho
        if (!this.savedAnchor)
        {
            // Anchor failed to save to the store.
            // Handle errors here.
        }
    }
}
```

Loading is essentially a reflection of the above:

```
private void LoadAnchor()
{
    this.savedAnchor = this.Load("MyAnchor", MyWorldAnchor);
    if (!this.savedAnchor)
    {
        // An anchor with that name wasn't saved to the store.
        // Handle errors here.
    }
}
```

To remove an anchor from the store, call the **Delete** method on the **WorldAnchorStore**. To remove all anchors from the store, call the **Clear** method.

HoloLens Anchor Sharing

Leave feedback

Anchor Sharing is a system for saving a World Anchor on one device in such a way that it can be loaded onto different devices.

For example: Two users play a game with a virtual game board placed on a table. For the two users' game boards to line up, both devices require a spatial understanding of where the virtual game board is placed, relative to the real world. Anchor Sharing allows Anchor information to be saved from one user's device and recreated on the other user's device.

Note that the Anchor Sharing feature does not contain the transport layer necessary to transfer data between devices. Network transport systems provide this functionality. See Unity documentation on Networking for more information on this.

Exporting Anchors

To export an existing World Anchor, you need a shared name for that Anchor and a `WorldAnchorTransferBatch`.

The example below shows basic usage of this API. Note that:

You should expect multiple calls to `OnExportDataAvailable`.
If you set up your application to send data to other devices incrementally, you also need to handle cases where data is sent but the export call fails mid-stream.

```
private void ExportWorldAnchor()
{
    WorldAnchorTransferBatch transferBatch = new WorldAnchorTransferBatch();
    transferBatch.AddWorldAnchor("GameRootAnchor", this.MyWorldAnchor);
    WorldAnchorTransferBatch.ExportAsync(transferBatch, OnExportDataAvailable, OnExport
}

private void OnExportComplete(SerializationCompletionReason completionReason)
{
    if (completionReason != SerializationCompletionReason.Succeeded)
    {
        // If we have been transferring data and it failed,
        // tell the client to discard the data
        SendExportFailedToClient();
    }
    else
    {
        // Tell the client that serialization has succeeded.
        // The client can start importing once all the data is received.
        SendExportSucceededToClient();
    }
}

private void OnExportDataAvailable(byte[] data)
{
    // Send the bytes to the client.  Data may also be buffered.
    TransferDataToClient(data);
}
```

Importing Anchors

Once the data is received, import it through `WorldAnchorTransferBatch` to then recreate the World Anchor on a different client.

The example below shows basic usage. Note that importing sometimes fails; if this happens, retry the process.

```
private int retryCount = 10;
private void ImportWorldAnchor(byte[] importedData)
{
    WorldAnchorTransferBatch.ImportAsync(importedData, OnImportComplete);
}

private void OnImportComplete(SerializationCompletionReason completionReason, WorldAnch
{
    if (completionReason != SerializationCompletionReason.Succeeded)
    {
        Debug.Log("Failed to import: " + completionReason.ToString());
        if (retryCount > 0)
        {
            retryCount--;
            WorldAnchorTransferBatch.ImportAsync(fileData, OnImportComplete);
        }
        return;
    }

    string[] ids = deserializedTransferBatch.GetAllIds();
    foreach (string id in ids)
    {
        GameObject gameObject = GetGameObjectFromAnchorId(id);
        if (gameObject != null)
        {
            transferBatch.LockObject(id, gameObject);
        }
        else
        {
            Debug.Log("Failed to find object for anchor id: " + id);
        }
    }
}
```

HoloLens Web Camera

Leave feedback

The Web **Camera** API is an asynchronous API that provides the ability to take pictures and record videos. The Web **Camera** is a shared resource. This means that only one `PhotoCapture` or `VideoCapture` instance can be activated at any given time. You can query what the current mode is using the WebCam.Mode property.

This section contains details of the **Web Camera** APIs and their capabilities:

Photo Capture
Video Capture

HoloLens photo capture

Leave feedback

Use the PhotoCapture API to take photos from the **HoloLens** web **camera**. You must enable the **WebCam** and **Microphone** capabilities to use the PhotoCapture API. The following example demonstrates how to take a photo using the PhotoCapture functionality and display it on a Unity GameObject.

```
using UnityEngine;
using System.Collections;
using System.Linq;
using UnityEngine.XR.WSA.WebCam;

public class PhotoCaptureExample : MonoBehaviour {
    PhotoCapture photoCaptureObject = null;
    Texture2D targetTexture = null;

    // Use this for initialization
    void Start() {
        Resolution cameraResolution = PhotoCapture.SupportedResolutions.OrderByD
        targetTexture = new Texture2D(cameraResolution.width, cameraResolution.h

        // Create a PhotoCapture object
        PhotoCapture.CreateAsync(false, delegate (PhotoCapture captureObject) {
            photoCaptureObject = captureObject;
            CameraParameters cameraParameters = new CameraParameters();
            cameraParameters.hologramOpacity = 0.0f;
            cameraParameters.cameraResolutionWidth = cameraResolution.width;
            cameraParameters.cameraResolutionHeight = cameraResolution.height;
            cameraParameters.pixelFormat = CapturePixelFormat.BGRA32;

            // Activate the camera
            photoCaptureObject.StartPhotoModeAsync(cameraParameters, delegate (P
                // Take a picture
                photoCaptureObject.TakePhotoAsync(OnCapturedPhotoToMemory);
            });
        });
    }

    void OnCapturedPhotoToMemory(PhotoCapture.PhotoCaptureResult result, PhotoCa
        // Copy the raw image data into the target texture
        photoCaptureFrame.UploadImageDataToTexture(targetTexture);

        // Create a GameObject to which the texture can be applied
        GameObject quad = GameObject.CreatePrimitive(PrimitiveType.Quad);
        Renderer quadRenderer = quad.GetComponent<Renderer>() as Renderer;
        quadRenderer.material = new Material(Shader.Find("Custom/Unlit/UnlitText
```

```
        quad.transform.parent = this.transform;
        quad.transform.localPosition = new Vector3(0.0f, 0.0f, 3.0f);

        quadRenderer.material.SetTexture("_MainTex", targetTexture);

        // Deactivate the camera
        photoCaptureObject.StopPhotoModeAsync(OnStoppedPhotoMode);
    }

    void OnStoppedPhotoMode(PhotoCapture.PhotoCaptureResult result) {
        // Shutdown the photo capture resource
        photoCaptureObject.Dispose();
        photoCaptureObject = null;
    }
}
```

Capture a photo to memory

When you capture an image to memory, a PhotoCaptureFrame is returned. A `PhotoCaptureFrame` contains both the native image data and spatial matrices that indicate where the image was taken.

Capturing an image to memory allows you to reference the captured image in a **Shader**, or apply it to a GameObject. There are three ways to extract the image data from the `PhotoCaptureFrame`. These are:

Access the image data as a Texture2D. This is the most common way to extract image data, because most **components** in Unity understand how to access image data in a Texture2D. Once your image has been captured to memory, you need to upload the image data into a Texture2D. Once the image data is uploaded into a Texture2D, you can then begin referencing that image data in materials, **scripts**, or any other relevant element of your project. The Unity API Documentation has an example which demonstrates how to capture a photo to memory and then upload it to a Texture2D. See WebCam.PhotoCapture.TakePhotoAsync to learn how to capture a photo to a Texture2D. Uploading the image data to a Texture2D via the upload command is the easiest way to start working with your image data in Unity. The upload operation happens on the main thread. This operation is resource-intensive and can affect the performance of your project.

Capture the native image data as a **WinRT IMFMediaBuffer**. See Microsoft's documentation on IMFMediaBuffer to learn more. Make a copy of the native image data by passing a byte list into the PhotoFrame.CopyRawImageDataIntoBuffer function. Please note that the copy operation occurs on the main thread. This operation is resource-intensive and can affect the performance of your project.

If you create your own **plugin**, or process the image data on a separate thread, you can get a pointer to the native image data via the PhotoFrame.GetUnsafePointerToBuffer function. The pointer returned is a pointer to an IMFMediaBuffer Component Object Model (COM). See Microsoft's documentation on IMFMediaBuffer and Component Object Models for more information. Once you call this function, a reference is added to the COM object. You are responsible for releasing the reference when you no longer need the resource.

HoloLens video capture

Leave feedback

Use the VideoCapture API to record videos directly to the file system in the MP4 format. Please note that the **VideoCapture** API does not work in the Unity Editor.

You must enable the WebCam and Microphone capabilities to use the **VideoCapture** API. The following example records a 5 second video and saves it to the file system.

```
using UnityEngine;
using System.Collections;
using System.Linq;
using UnityEngine.XR.WSA.WebCam;

public class VideoCaptureExample : MonoBehaviour
{
    static readonly float MaxRecordingTime = 5.0f;

    VideoCapture m_VideoCapture = null;
    float m_stopRecordingTimer = float.MaxValue;

    // Use this for initialization
    void Start()
    {

        StartVideoCaptureTest();
    }

    void Update()
    {
        if (m_VideoCapture == null || !m_VideoCapture.IsRecording)
        {
            return;
        }

        if (Time.time > m_stopRecordingTimer)
        {
            m_VideoCapture.StopRecordingAsync(OnStoppedRecordingVideo);
        }
    }

    void StartVideoCaptureTest()
    {

        Resolution cameraResolution = VideoCapture.SupportedResolutions.OrderByD
        Debug.Log(cameraResolution);
```

```
    float cameraFramerate = VideoCapture.GetSupportedFrameRatesForResolution
    Debug.Log(cameraFramerate);

    VideoCapture.CreateAsync(false, delegate (VideoCapture videoCapture)
    {
        if (videoCapture != null)
        {
            m_VideoCapture = videoCapture;
            Debug.Log("Created VideoCapture Instance!");

            CameraParameters cameraParameters = new CameraParameters();
            cameraParameters.hologramOpacity = 0.0f;
            cameraParameters.frameRate = cameraFramerate;
            cameraParameters.cameraResolutionWidth = cameraResolution.width;
            cameraParameters.cameraResolutionHeight = cameraResolution.heigh
            cameraParameters.pixelFormat = CapturePixelFormat.BGRA32;

            m_VideoCapture.StartVideoModeAsync(cameraParameters,
                                               VideoCapture.AudioState.Appli
                                               OnStartedVideoCaptureMode);
        }
        else
        {
            Debug.LogError("Failed to create VideoCapture Instance!");
        }
    });
}

void OnStartedVideoCaptureMode(VideoCapture.VideoCaptureResult result)
{
    Debug.Log("Started Video Capture Mode!");
    string timeStamp = Time.time.ToString().Replace(".", "").Replace(":", ""
    string filename = string.Format("TestVideo_{0}.mp4", timeStamp);
    string filepath = System.IO.Path.Combine(Application.persistentDataPath,
    filepath = filepath.Replace("/", @"\");
    m_VideoCapture.StartRecordingAsync(filepath, OnStartedRecordingVideo);
}

void OnStoppedVideoCaptureMode(VideoCapture.VideoCaptureResult result)
{
    Debug.Log("Stopped Video Capture Mode!");
}

void OnStartedRecordingVideo(VideoCapture.VideoCaptureResult result)
{
    Debug.Log("Started Recording Video!");
```

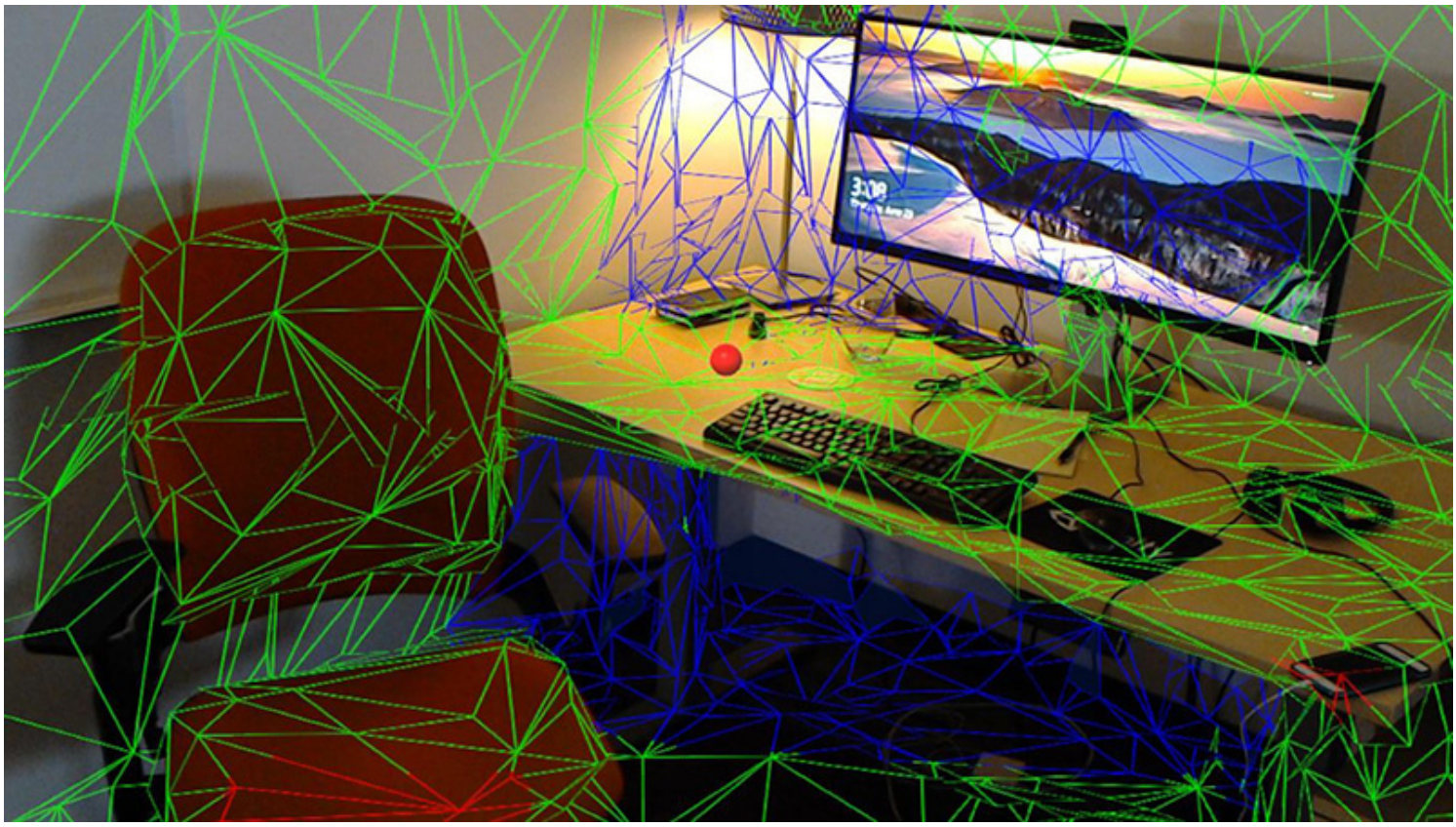
```
        m_stopRecordingTimer = Time.time + MaxRecordingTime;
    }

    void OnStoppedRecordingVideo(VideoCapture.VideoCaptureResult result)
    {
        Debug.Log("Stopped Recording Video!");
        m_VideoCapture.StopVideoModeAsync(OnStoppedVideoCaptureMode);
    }
}
```

HoloLens Spatial Mapping

Leave feedback

HoloLens has built-in **cameras** that continuously scan the environment, allowing it to construct virtual world geometry for real-world objects. This ability allows **HoloLens** to create convincing interactions between virtual content and the real world (such as occluding and physically interacting with real-world objects). The process of mapping real-world surfaces into the virtual world is called **Spatial Mapping**.

The Unity Editor has a low-level Scripting API for gathering information about Surfaces in your project environment. This API gives you maximum control over when to query the device for Surface changes, and when to create or update the corresponding Surface game objects. Unity's Spatial Mapping **components** allow you to quickly get up and running with **Windows Mixed Reality**, without directly using the low-level Scripting API.



An example of the Spatial Mapping feature applied to a real-world space

Spatial Mapping concepts

Spatial Mapping is a great way to add better real-world interaction to your HoloLens application.

Mapping

The HoloLens device constantly scans its surroundings and refines its understanding of the world based on incoming sensor data. The device frequently updates that data so that the device can pick up environmental changes, such as people moving through the room or doors opening and closing. Spatial Mapping saves the world mapping data set to the HoloLens device and this data persists across multiple applications and device restarts.

Transparent, black, and reflective Surfaces do not work well with Spatial Mapping on the HoloLens. If the device cannot detect a real-world object, an empty patch appears in the Spatial Mapping data. The same is true for parts of the world that the device has not been in or cannot see. For example, no data exists for rooms that the HoloLens has not observed.

Data organization

The device's world mapping is divided into uniform chunks called Surfaces. The Spatial Mapping system orients Surfaces in the world in a way convenient to the system itself. There is no guarantee that the system will arrange Surfaces in any particular orientation, and Surfaces may also improperly intersect a given real-world space (such as a room). When Spatial Mapping generates data for a Surface, the data slightly overlaps with neighboring Surfaces.

Note: There is no semantic meaning or interpretation associated with any of the Surface data. Spatial Mapping does not know and cannot report on what is actually on a Surface. For example, it can't tell that what it detects as a blob on a desk is actually a mug, or what the chair-shaped object in the middle of the room is. It only reports on the configuration of the geometry in that area based on its understanding of the world, which it constructs using its sensory input.

Spatial Mapping important considerations

To avoid performance issues when developing applications using Spatial Mapping, consider the following:

The amount of Spatial Mapping data can be very large, which can cause scalability challenges.

Objects or people moving quickly around the room can make Spatial Mapping data very irregular, so you should avoid this where possible.

Holes in the data can sometimes cause issues, particularly if you need continuous data for design reasons. These holes are most commonly caused by transparent or black surfaces.

See Microsoft's documentation on Spatial Mapping to learn more about Spatial Mapping concepts.

2018–05–01 Page published with editorial review

Spatial Mapping for Hololens documentation updated in 2017.3

Spatial Mapping components

Leave feedback

To facilitate **Spatial Mapping** in Unity and allow easy access to its features, Unity provides two **Spatial Mapping XR** components:

Spatial Mapping Renderer

Spatial Mapping Collider

You can use these components together or independently. Each Spatial Mapping component uses its own `SurfaceObserver` to understand changes to the physical world. Each Spatial Mapping component periodically queries the Spatial Mapping system to understand what changes have occurred in the physical environment (how often they perform these queries depends on how you configure them). When the system notifies the component of relevant changes, the component prioritizes baking the various changed Surfaces.

The baking process involves generating a **Mesh Filter** with a **Mesh** that corresponds to the physical Surface. The components use this **Mesh Filter** in their own specific ways.

Each Spatial Mapping component is independent of other Spatial Mapping components. This means that each component maintains its own list of Surfaces, even if multiple components see the same Surface. We recommend that you limit the number of Spatial Mapping components you use, in order to optimize performance.

2018–05–01 Page published with editorial review

Spatial Mapping for Hololens documentation updated in 2017.3

SpatialMapping Components General Settings

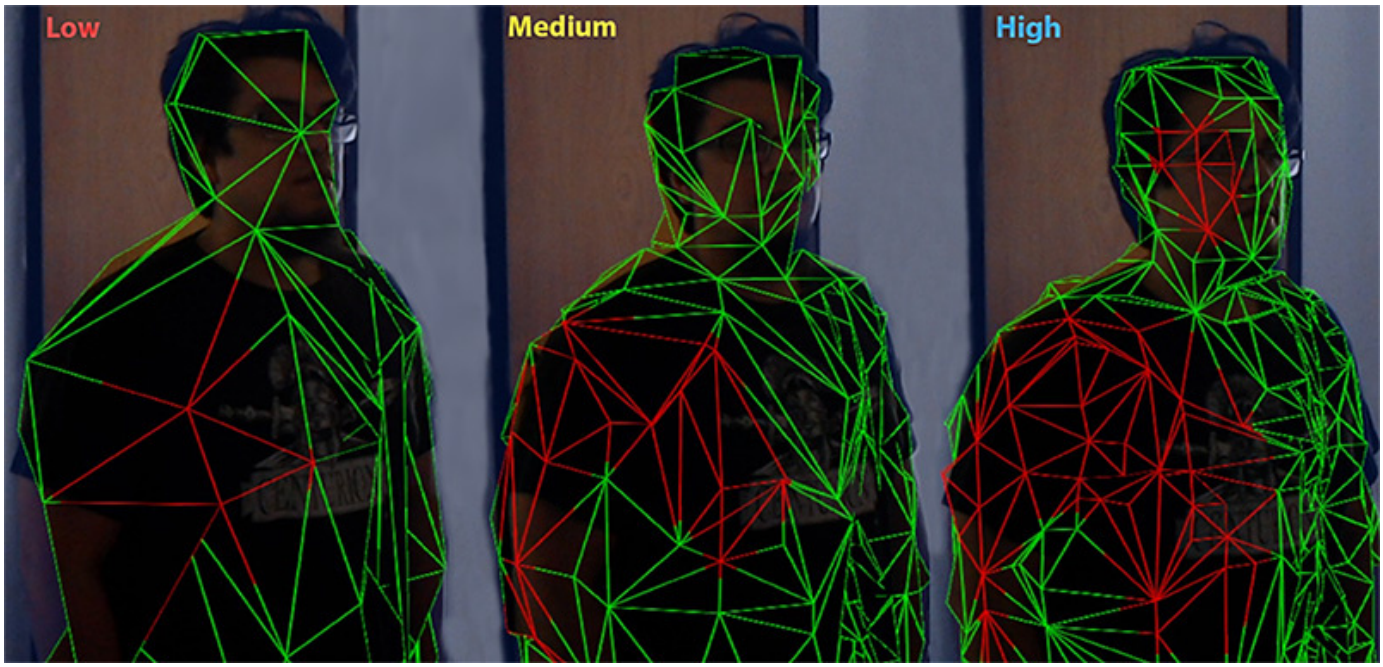
Leave feedback

The settings below are common to both **Spatial Mapping Renderer** and **Spatial Mapping Collider components**.

Setting	Description
Surface Parent	Select the **Surface Parent** GameObject that you want Surface **GameObjects** generated by **Spatial Mapping** components to inherit from. Leave this as **None(Game Object)**, to automatically generate a Surface Parent GameObject.
Freeze Updates	Check this box to stop the component querying the system for Surface changes. **Note:** Each Spatial Mapping component periodically queries the Spatial Mapping data for Surface changes in physical space. Querying and baking Surfaces costs memory, performance, and power. For environments that you expect to be mostly static, we recommend that you allow users to look around the environment for a duration of time without updating the Surface GameObjects. If you expect the environment in your simulation to be mostly static and unchanging (like a board game), you can scan as much Surface data as you need when your application starts, and then set the **Freeze Updates** property to **true** to prevent further updates. This increases performance slightly and consumes less power.
Time Between Updates	The time in decimal format seconds (for example, 3.7 or 4.6) between queries for Surface changes in physical space. The default is 2.5 seconds. Note that the more regular the queries, the higher the cost in memory, performance, and power.
Removal Update Count	The number of updates before a Surface GameObject is removed by the system. You can think of an update as a frame in this case. The default is 10 updates. **Note**: The removal update countdown begins when Spatial Mapping notifies the component that a Surface GameObject is no longer in the SurfaceObserver's bounding volume (in that it is no longer within the defined area that the system reports on). This setting allows you to specify the number of updates that should happen after this event before Spatial Mapping removes the Surface GameObject.
Level of Detail	The quality of the **Mesh** that the component generates (**Low**, **Medium**, or **High**). The default quality is **Medium**. The higher the quality, the more refined and accurate the generated **Collider** or rendered Mesh. Using lower quality settings results in a lower cost in performance and power consumption. See the image below this table for an example of the three **Level Of Detail** modes.
Bounding Volume Type	The component's bounding volume area shape, in which the application receives Spatial Mapping data. This can be either a **Sphere** or **Axis Aligned Box**. The default is **Axis Aligned Box**. **Note:** The bounding volume is the defined area about which the system reports physical Surface changes and limits the extents of the Spatial Mapping Mesh.
Size In Meters	The size of the bounding volume used by component (in meters). Configure **Sphere** by radius; the default radius is 2 meters. Configure **Axis Aligned Box** by its extents; the default is a Vector3 (4,4,4). **Note:** The observer's bounding volume is the defined area about which Spatial Mapping reports physical Surface changes.

Level of Detail

Both Spatial Mapping components allow you to specify one of three Levels of Detail for each component's generated spatial meshes (Low, Medium, or High) as depicted in the image below.



The three Level Of Detail modes for Spatial Mapping meshes

When possible, set the Level of Detail setting for **Spatial Mapping Colliders** to Low. This increases performance and reduces power usage when calculating **collision** intersections. However, setting **Level of Detail** to **Low** can negatively affect the visual effect of your application,

Spatial Mapping Collider

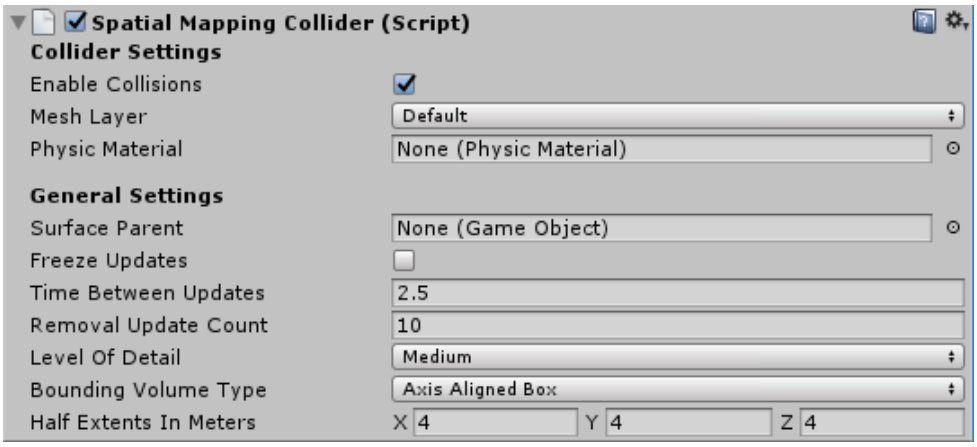
Leave feedback

The **Spatial Mapping Collider component** allows **holographic** content to interact with real-world physical Surfaces. This **component** handles creating, updating, and destroying the Surface **GameObject Colliders** in the **Scene**.

The component periodically queries the system for Surface changes in the physical world. When the system reports Surface changes, the **Spatial Mapping Collider** component prioritizes when each reported Surface is baked by Unity. Every time a Surface is baked by the system, a new GameObject is then generated by Unity, containing a Mesh Filter and Mesh Collider component. A Surface with a **Mesh** Collider allows raycasts to collide with it as with any other **mesh** in Unity. See Camera Raycast documentation for more information.

Note: This component only updates the Mesh Collider components of Surface GameObjects, and not the **Mesh Renderers**.

Note: Spatial Mapping__ Colliders__ update with less latency than **Spatial Mapping Mesh Renderers**. This means that Colliders update faster than Renderers.



Spatial Mapping Collider component as it appears in the Unity Editor

Collider Settings

Setting	Description
Enable Collisions	Check this box to enable Surface Mesh Colliders. This means that holographic content can collide with Surfaces.
Mesh Layer	Set the Layer property on all the Surface Mesh Colliders. Note that you need to set Layers for raycasts. When performing a raycast, you must indicate which Layers you want the ray intersection to test against. See Camera Raycast documentation for more information, and see Example script: SpatialSurface raycast, below. By default, Unity assigns all GameObjects to the **Default** Layer. However, it is good practice to assign your GameObjects to a specific Layer. We recommend that you create a new layer specifically for Spatial Mapping Surfaces and assign this Layer to the Mesh Layer property.
Physic Material	Specify which **Physic Material** to assign to each Surface GameObject's Mesh Collider. The default is **None (Physic Material)**. A **Physic Material** specifies how other Rigidbody components should interact with it. For example, you might want to create a Surface that simulates ice, and therefore applies less friction to a GameObject moving on it. The Spatial Mapping Collider component applies its **Physic Material** to all of the Mesh Colliders on its Surface GameObjects. See documentation on Physic Material for more information

General Settings

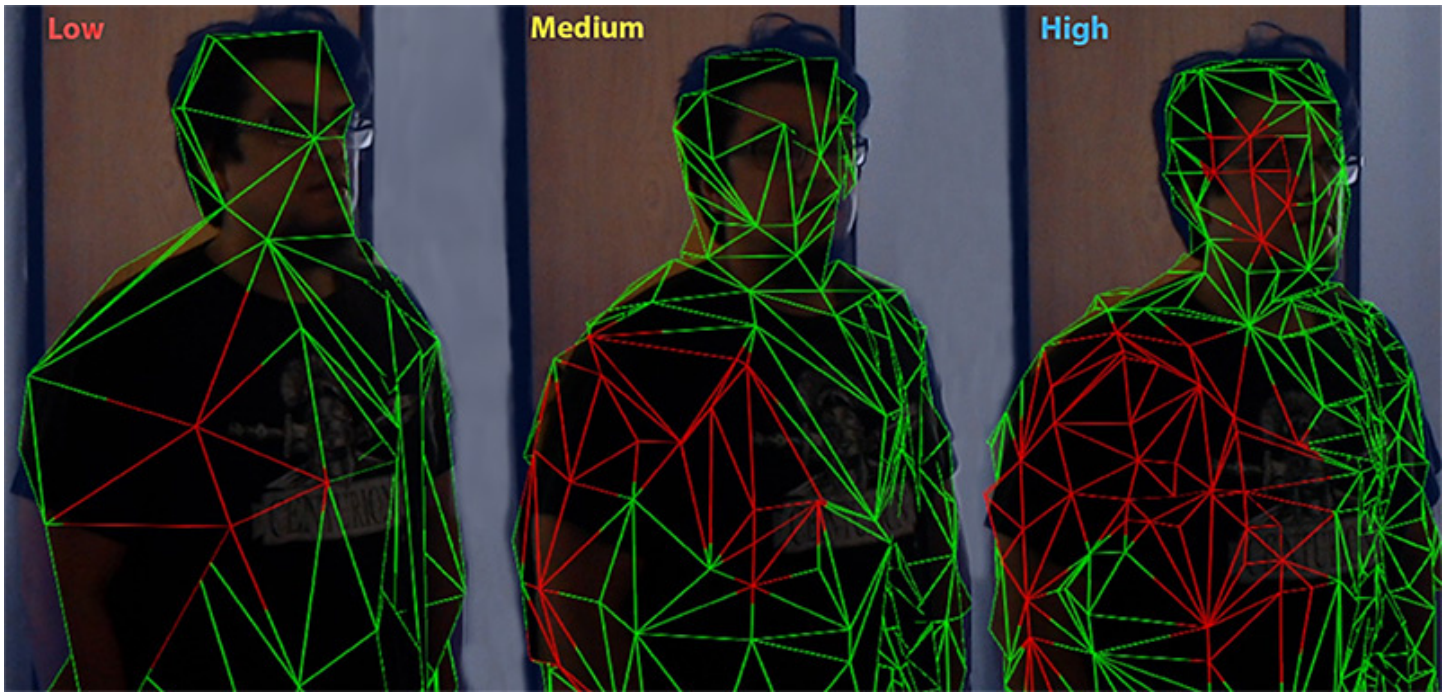
SpatialMapping Components General Settings

The settings below are common to both **Spatial Mapping Renderer** and **Spatial Mapping Collider** components.

Setting	Description
Surface Parent	Select the **Surface Parent** GameObject that you want Surface GameObjects generated by Spatial Mapping components to inherit from. Leave this as **None(Game Object)**, to automatically generate a Surface Parent GameObject.
Freeze Updates	Check this box to stop the component querying the system for Surface changes. **Note:** Each Spatial Mapping component periodically queries the Spatial Mapping data for Surface changes in physical space. Querying and baking Surfaces costs memory, performance, and power. For environments that you expect to be mostly static, we recommend that you allow users to look around the environment for a duration of time without updating the Surface GameObjects. If you expect the environment in your simulation to be mostly static and unchanging (like a board game), you can scan as much Surface data as you need when your application starts, and then set the **Freeze Updates** property to **true** to prevent further updates. This increases performance slightly and consumes less power.
Time Between Updates	The time in decimal format seconds (for example, 3.7 or 4.6) between queries for Surface changes in physical space. The default is 2.5 seconds. Note that the more regular the queries, the higher the cost in memory, performance, and power.
Removal Update Count	The number of updates before a Surface GameObject is removed by the system. You can think of an update as a frame in this case. The default is 10 updates. **Note**: The removal update countdown begins when Spatial Mapping notifies the component that a Surface GameObject is no longer in the SurfaceObserver's bounding volume (in that it is no longer within the defined area that the system reports on). This setting allows you to specify the number of updates that should happen after this event before Spatial Mapping removes the Surface GameObject.
Level of Detail	The quality of the Mesh that the component generates (**Low**, **Medium**, or **High**). The default quality is **Medium**. The higher the quality, the more refined and accurate the generated Collider or rendered Mesh. Using lower quality settings results in a lower cost in performance and power consumption. See the image below this table for an example of the three **Level Of Detail** modes.
Bounding Volume Type	The component's bounding volume area shape, in which the application receives Spatial Mapping data. This can be either a **Sphere** or **Axis Aligned Box**. The default is **Axis Aligned Box**. **Note:** The bounding volume is the defined area about which the system reports physical Surface changes and limits the extents of the Spatial Mapping Mesh.
Size In Meters	The size of the bounding volume used by component (in meters). Configure **Sphere** by radius; the default radius is 2 meters. Configure **Axis Aligned Box** by its extents; the default is a Vector3 (4,4,4). **Note:** The observer's bounding volume is the defined area about which Spatial Mapping reports physical Surface changes.

Level of Detail

Both Spatial Mapping components allow you to specify one of three Levels of Detail for each component's generated spatial meshes (Low, Medium, or High) as depicted in the image below.



The three Level Of Detail modes for Spatial Mapping meshes

When possible, set the Level of Detail setting for **Spatial Mapping Colliders** to Low. This increases performance and reduces power usage when calculating **collision** intersections. However, setting **Level of Detail** to **Low** can negatively affect the visual effect of your application,

Using the Spatial Mapping Collider

Surface GameObjects

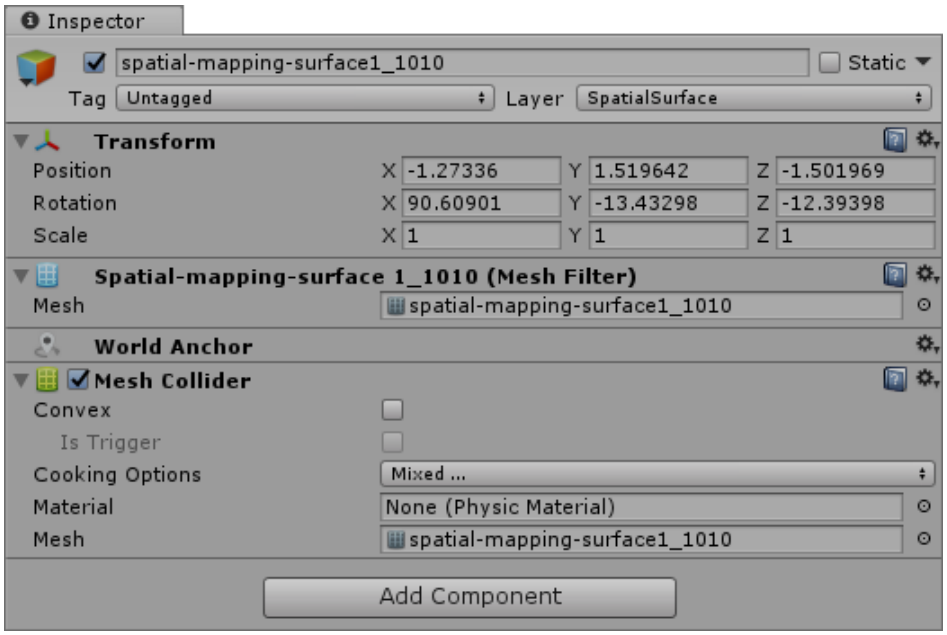
When you add a **Spatial Mapping Collider** component to a GameObject in your Scene, it generates its own separate set of Surface GameObjects during runtime. These appear in your Scene parented to a Surface Parent GameObject.

The Surface GameObjects generated by the **Spatial Mapping Collider** contain the following components:

Mesh Filter

World Anchor

Mesh Collider



Components of a Surface GameObject generated by a **Spatial Mapping Collider**

If you add multiple **Spatial Mapping Collider** components to your Scene, each individual component generates its own set of Surface GameObjects. For example, if two GameObjects in your Scene contain a **Spatial Mapping Collider** then your Scene contains two sets of Surface GameObjects; one generated by each component at runtime. This is important to keep in mind for optimization purposes. s **Note:** When first generated by the Spatial Mapping system, all **Surface GameObjects** are assigned to the **Default Layer**. However, it is good practice to assign your GameObjects to a specific Layer, because raycasting is an expensive calculation to perform. By using Layers, you can filter which GameObjects you are doing your raycast calculations against, and so optimise performance. If you don't have a lot of complicated Meshes on your **Default** Layer, then doing a raycast test for collision won't have a large performance cost. However, it is best to organize your GameObjects into Layers to reduce the complexity of raycast tests when doing collisions. You can change the layer assigned to all Surface GameObjects generated by a specific **Spatial Mapping Collider** component by changing the **Collider Settings** > **Mesh Layer** property.

Example script: SpatialSurface raycast on a tap gesture event

The following example demonstrates how to raycast against GameObjects on the **SpatialSurface** Layer and use a GestureRecognizer to initiate this raycast upon detection of a tap gesture event.

```
using UnityEngine;
using System.Collections;

//Required for Gesture Recognizer
using UnityEngine.XR.WSA.Input;


public class CustomLayerCollision : MonoBehaviour
{

    //use this to capture gesture events

    private GestureRecognizer GR_recognizer;

    void Start()
    {
        //Initialise the GestureRecognizer
```

```
        GR_recognizer = new GestureRecognizer();

        //Initiate gesture capture

        GR_recognizer.StartCapturingGestures();

        //Tell the GestureRecognizer which events to listen for

        GR_recognizer.SetRecognizableGestures(GestureSettings.Tap);

        //Subscribe the Tapped event to the DetectCollisions method

        //This makes this method be called any time the GestureRecognizer detects a tap ev

        GR_recognizer.Tapped += DetectCollisions;

        //When the user initiates a tap gesture, the DetectCollisions

        //method is called by the Tapped event delegate to do a collision test using Rayca

        //See [GestureRecognizer API reference](../ScriptReference/XR.WSA.Input.GestureRec
    }
    //Method to fire a raycast on a Tap event
    public void DetectCollisions(TappedEventArgs tapEvent)
    {
        // Raycast against all GameObjects that are on either the
        // spatial Surface or UI layers.

        int layerMask = 1 << LayerMask.NameToLayer("SpatialSurface");

        //construct a Ray using the position and forward direction of the User’s head
        Ray GazeRay = new Ray(tapEvent.headPose.position, tapEvent.headPose.forward);

        //Raycast using constructed Ray from Gaze and store collisions in array hits

        RaycastHit[] hits = Physics.RaycastAll(GazeRay, float.MaxValue, layerMask);

        if (hits.Length > 0)
        {
            foreach (RaycastHit hit in hits)
            {
                Debug.Log(string.Format("Hit Object **\"**{0}**\"** at position **\"**{1}*
            }
        }
        else
        {
            Debug.Log("Nothing was hit.");
        }
    }
}
```

See documentation on Layers and Raycasting from Camera for more information.

2018–05–01 Page published with editorial review

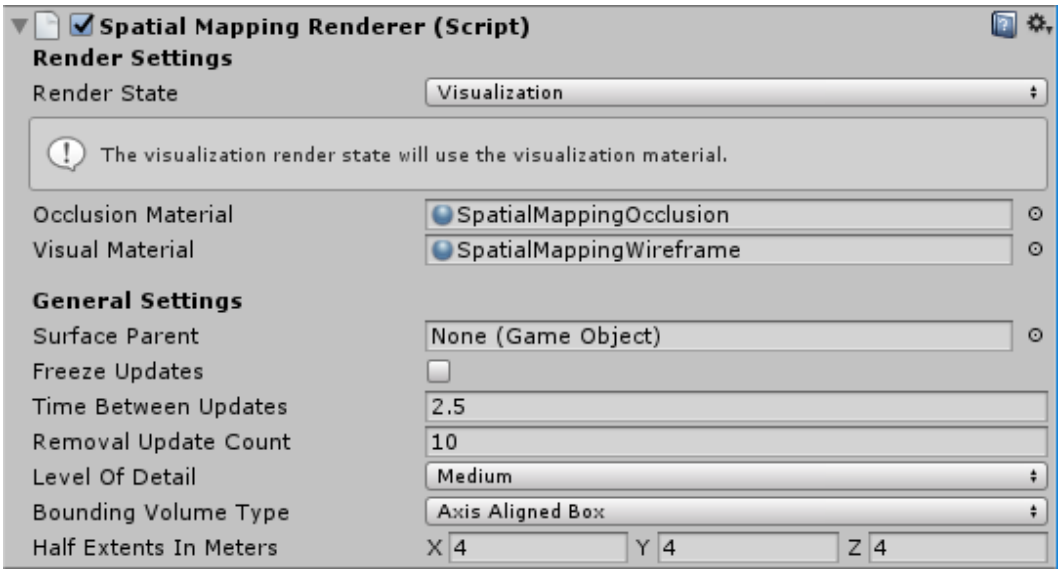
Spatial Mapping for Hololens documentation updated in 2017.3

Spatial Mapping Renderer

Leave feedback

The **Spatial Mapping Renderer component** provides a visual representation of **Spatial Mapping** Surfaces. This is useful for visually debugging Surfaces and adding visual effects to the environment.

The system periodically sends data to the **Spatial Mapping Renderer** component about changes in physical space. Each time the component is notified of these changes by the Spatial Mapping system, it bakes the returned Surface data into **GameObjects** that contain both Mesh Filter and Mesh Renderer components. The Spatial Mapping Renderer component manages the lifetime of these Surface GameObjects. This means that it handles creating, updating, and destroying the Surface GameObject **Mesh** Renderers in the **Scene**.



Spatial Mapping Renderer component as it appears in the Unity Editor

Render Settings

The table below lists all unique Render Settings available for the **Spatial Mapping Renderer** component.

Setting	Description
Render State	All Surface GameObjects take their material from the **Render State** setting. When you change the **Render State** setting (for example, by script during run time), all Surface GameObjects' render Materials change to those of the chosen **Render State** setting. This reduces the number of draw calls, which in turn improves the rendering performance. Using a shared material also reduces the amount of memory used by the application during rendering. Select one of the three options to render Surfaces. See Render State, below, for more information.
Occlusion	Apply the Material defined in **Occlusion Material**. This is a transparent Material which hides GameObjects behind real world Surfaces. See Visual and Occlusion Materials, below, for more information. **Note:** This enables all of a Surface's Mesh Renderers, overriding any other setting.

Setting	Description
Visualisation	Apply the Material defined in **Visualization Material** for visualizing the Spatial Mapping Surfaces (usually a wireframe Material) in your environment. See Visual and Occlusion Materials, below, for more information. **Note**: This enables all of a Surface's Mesh Renderers, overriding any other setting.
None	Choose to disable all the Mesh Renderers assigned to the Spatial Mapping Surfaces.
Occlusion Material	The material you select here applies when the Spatial Mapping Renderer's **Render State** is set to **Occlusion**. The default is the built-in **SpatialMappingOcclusion** material.
Visual Material	The material you select here is the Visual Material that applies when the Spatial Mapping Renderer **Render State** is set to **Visualization**. The default is the built-in **SpatialMappingWireframe** material.

General Settings

SpatialMapping Components General Settings

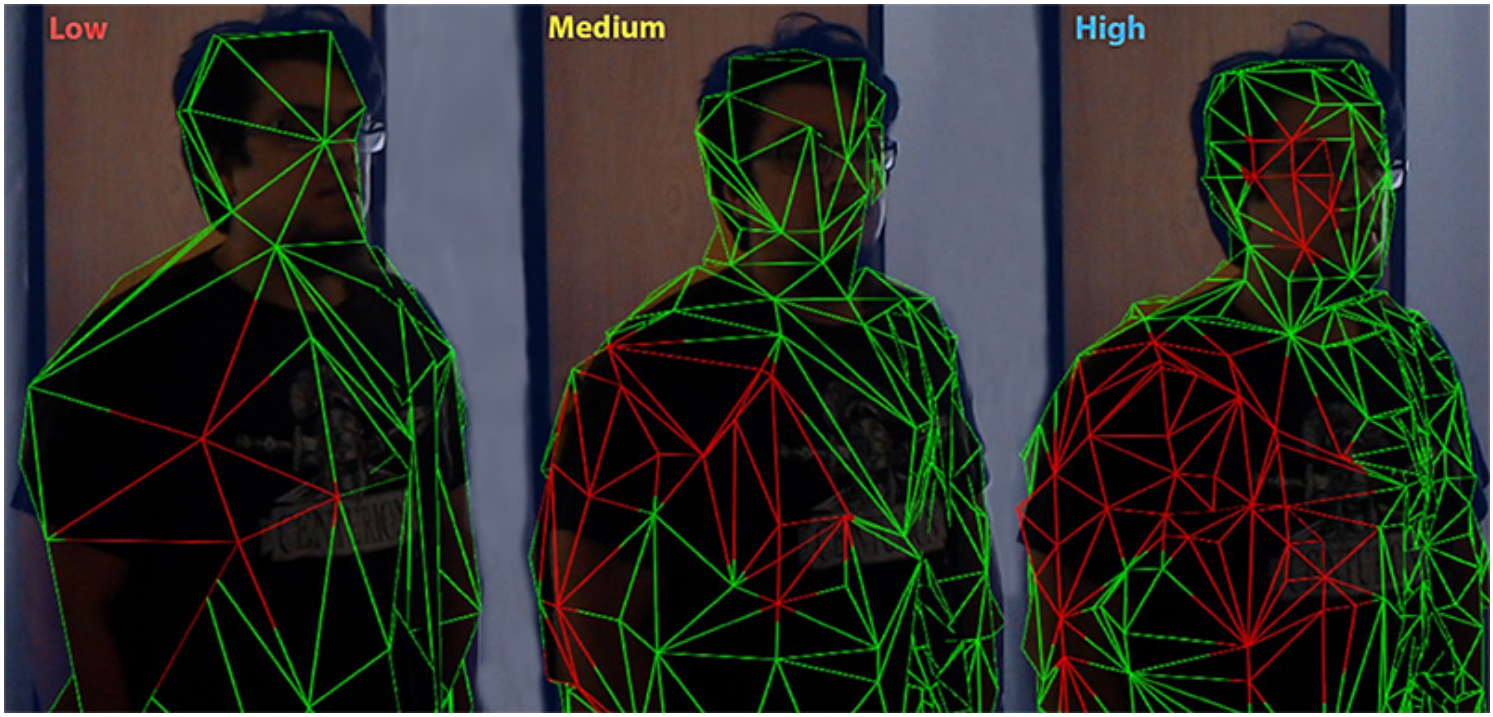
The settings below are common to both **Spatial Mapping Renderer** and **Spatial Mapping Collider** components.

Setting	Description
Surface Parent	Select the **Surface Parent** GameObject that you want Surface GameObjects generated by Spatial Mapping components to inherit from. Leave this as **None(Game Object)**, to automatically generate a Surface Parent GameObject.
Freeze Updates	Check this box to stop the component querying the system for Surface changes. **Note:** Each Spatial Mapping component periodically queries the Spatial Mapping data for Surface changes in physical space. Querying and baking Surfaces costs memory, performance, and power. For environments that you expect to be mostly static, we recommend that you allow users to look around the environment for a duration of time without updating the Surface GameObjects. If you expect the environment in your simulation to be mostly static and unchanging (like a board game), you can scan as much Surface data as you need when your application starts, and then set the **Freeze Updates** property to **true** to prevent further updates. This increases performance slightly and consumes less power.
Time Between Updates	The time in decimal format seconds (for example, 3.7 or 4.6) between queries for Surface changes in physical space. The default is 2.5 seconds. Note that the more regular the queries, the higher the cost in memory, performance, and power.
Removal Update Count	The number of updates before a Surface GameObject is removed by the system. You can think of an update as a frame in this case. The default is 10 updates. **Note**: The removal update countdown begins when Spatial Mapping notifies the component that a Surface GameObject is no longer in the SurfaceObserver's bounding volume (in that it is no longer within the defined area that the system reports on). This setting allows you to specify the number of updates that should happen after this event before Spatial Mapping removes the Surface GameObject.

Setting	Description
Level of Detail	The quality of the Mesh that the component generates (**Low**, **Medium**, or **High**). The default quality is **Medium**. The higher the quality, the more refined and accurate the generated **Collider** or rendered Mesh. Using lower quality settings results in a lower cost in performance and power consumption. See the image below this table for an example of the three **Level Of Detail** modes.
Bounding Volume Type	The component's bounding volume area shape, in which the application receives Spatial Mapping data. This can be either a **Sphere** or **Axis Aligned Box**. The default is **Axis Aligned Box**. **Note:** The bounding volume is the defined area about which the system reports physical Surface changes and limits the extents of the Spatial Mapping Mesh.
Size In Meters	The size of the bounding volume used by component (in meters). Configure **Sphere** by radius; the default radius is 2 meters. Configure **Axis Aligned Box** by its extents; the default is a Vector3 (4,4,4). **Note:** The observer's bounding volume is the defined area about which Spatial Mapping reports physical Surface changes.

Level of Detail

Both Spatial Mapping components allow you to specify one of three Levels of Detail for each component's generated spatial meshes (Low, Medium, or High) as depicted in the image below.



The three Level Of Detail modes for Spatial Mapping meshes

When possible, set the Level of Detail setting for **Spatial Mapping Colliders** to Low. This increases performance and reduces power usage when calculating **collision** intersections. However, setting **Level of Detail** to **Low** can negatively affect the visual effect of your application,

Using the Spatial Mapping Renderer

Render State

This component provides an easy way to change the material on all generated Surfaces dynamically. Unity provides two pre-built Material types for you to use:

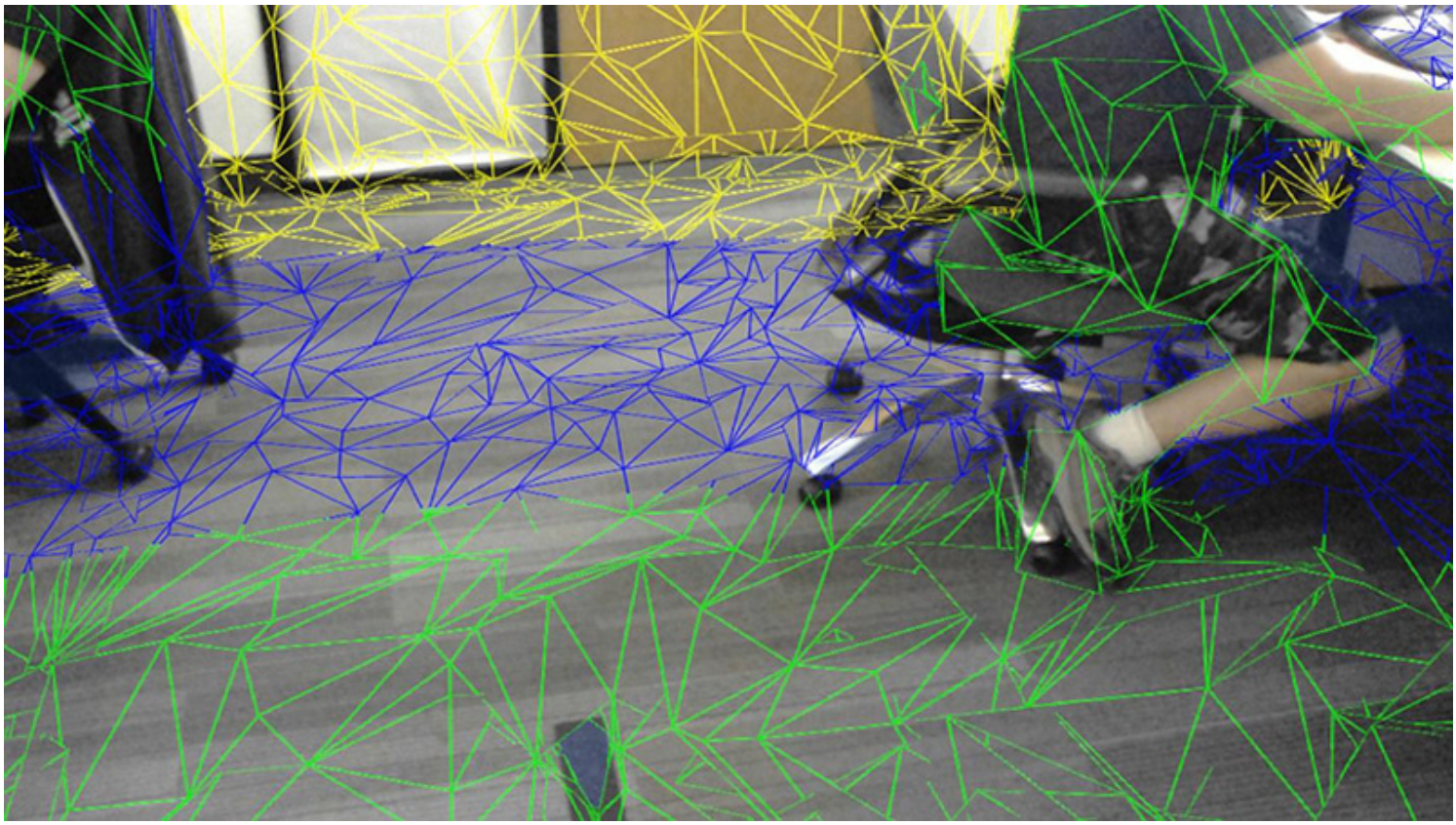
Occlusion Material

This makes the GameObject appear transparent, but holograms are not visible through the GameObject. For example, this might be useful if you require a real-world desk to conceal an in-game **holographic** object placed underneath it.

Visual Material

This is a wireframe Material that Unity applies to all Surfaces in the Spatial Mapping Renderer component. With the default Wireframe **Shader**, the colors of the wireframe represent real-world distances. This is usually the most useful option for debugging, but you can also use it for visual effects. The **Shader** uses color mapping to indicate distance; the table below shows this color mapping:

Distance from HoloLens	Color
0 to 1 meters	Black
1 to 2 meters	Red
2 to 3 meters	Green
3 to 4 meters	Blue
4 to 5 meters	Yellow
5 to 6 meters	Cyan
6 to 7 meters	Magenta
7 to 8 meters	Maroon
8 to 9 meters	Teal
9 to 10 meters	Orange
10 meters or greater	White



The default wireframe Shader used as the Visualization Material in the **Spatial Mapping Renderer** component (colors in the wireframe represent distances)

When you assign a new Material to either the **Visual Material** or **Occlusion Material,** the Material of your Surface GameObjects does not automatically change. To apply the new Material to all Surfaces, you need to set the **Render State** to the Material you have changed (**Visual Material** or **Occlusion Material)**.

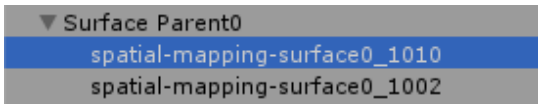
If you only assign the **Occlusion Material** or **Visual Material** properties in **Render State** from the Unity Editor (directly on the GameObject in your scene), Spatial Mapping destroys them at the same time as the **Spatial Mapping Renderer** component. However, Spatial Mapping doesn't destroy any **Occlusion** or **Visual Materials** assigned or changed through script with the component, so you will need to destroy them manually.

The code snippet below changes the Material that Spatial Mapping applies to all Surface GameObjects dynamically at run time.

```
SpatialMappingRenderer renderer = spatialMappingGameObject.AddComponent<SpatialMap
renderer.visualMaterial = new Material(Shader.Find("VR/SpatialMapping/Wireframe"))
renderer.renderState = SpatialMappingRenderer.RenderState.Visualization;
```

Surface GameObjects

The **Spatial Mapping Renderer** component creates its own separate set of Surface GameObjects during run time. These appear in your Scene parented to a Surface Parent GameObject:



Example Surface Parent GameObject and Surface GameObject children

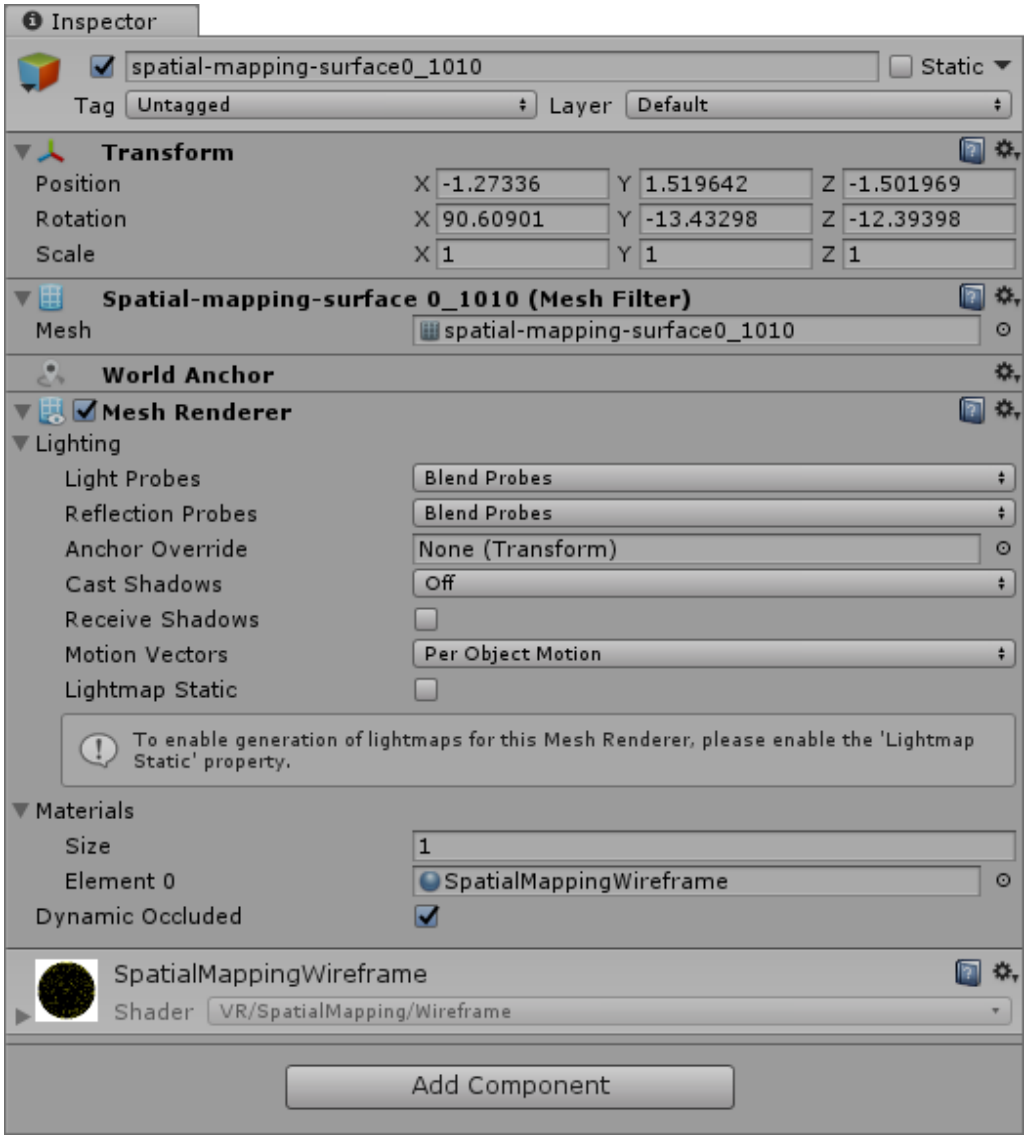
When a **Spatial Mapping Renderer** generates Surface GameObjects, they contain the following component

Mesh Filter

World Anchor

Mesh Renderer

A shared Material (this is either a Visualization or Occlusion Material, depending on the Render State setting)



Inspector view of a Surface GameObject generated by a **Spatial Mapping Renderer**

If you add multiple **Spatial Mapping Renderer** components to your Scene, each component generates its own set of Surface GameObjects. For example, if you have two GameObjects with **Spatial Mapping Renderer** components, then your Scene contains two sets of Surface GameObjects: one generated by each component at run time. This is important to keep in mind for optimization purposes.

2018–05–01 Page published with editorial review

Spatial Mapping for Hololens documentation updated in 2017.3

Spatial Mapping low level API

Leave feedback

The **Spatial Mapping** Renderer and **Collider components** allow you to use the features of **Spatial Mapping** easily without worrying about the finer details of the system. If you want to have finer control over the **Spatial Mapping** in your application then use the low level API that Unity provides for **Spatial Mapping**.

The API provides a number of data structures and object types for accessing the Spatial Mapping information that the **HoloLens** gathers.

SurfaceObserver

You can access Spatial Mapping data using a SurfaceObserver . A `SurfaceObserver` is an API class which monitors a volume of real-world space for which the application requires Spatial Mapping data. The `SurfaceObserver` describes a physical area in the real-world and reports on the set of spatial Surfaces it intersects with that have been added, changed, or removed by the Spatial Mapping system.

Use of `SurfaceObservers` is only necessary if you wish to interact with Spatial Mapping directly through the Unity scripting API.

Unity provides its own Spatial Mapping Renderer and Collider components, built on the `SurfaceObserver` API to allow easy access to Spatial Mapping functionality. See documentation on Spatial Mapping components for more details on these.

Using `SurfaceObserver`, applications can asynchronously request **Mesh** data with or without physics **collision** data. When the request is complete, another callback informs the application that the data is ready to use.

`SurfaceObserver` provides the following functionality to your application:

Issues callbacks upon request for Surface changes such as additions, removals, and updates.

Provides an interface for requesting Mesh data corresponding to a known Surface.

Issues callbacks when the Mesh data it requests is ready for use.

Provides ways of defining the location and volume of the `SurfaceObserver`.

SurfaceData

SurfaceData is a class which contains all the information required by the Spatial Mapping system to build and report on a **Surface's** Mesh data.

You must pass a populated `SurfaceData` object to the Spatial Mapping system using the RequestMeshAsync method. When you initially call the RequestMeshAsync method, you need to pass it a `SurfaceDataReadyDelegate`. When the Mesh data is ready the `SurfaceDataReadyDelegate` reports a matching `SurfaceData` object.

This allows the application to determine precisely which Surface the data corresponds to.

You should populate the `SurfaceData` GameObject using the information your application requires. This includes the following components and data:

A WorldAnchor component

A MeshFilter component

A MeshCollider component (if physics data is required in your application)

The triangles per cubic meter of the generated Mesh that you want

The Surface ID

The system throws argument exceptions when you call the RequestMeshAsync method with an incorrectly configured SurfaceData object. Even if a `RequestMeshAsync` method call does not throw argument exceptions, there is no other way to check whether Spatial Mapping is creating and returning Mesh data successfully. We recommend that you keep track of the Mesh data you create manually through script.

API usage example

The sample script below shows basic examples of using the important parts of the API.

```
using UnityEngine;
using UnityEngine.__XR__;
using UnityEngine.XR.WSA;
using UnityEngine.__Rendering__;
using UnityEngine.Assertions;
using System;
using System.Collections;
using System.Collections.Generic;

public enum BakedState
{
    NeverBaked = 0,
    Baked = 1,
    UpdatePostBake = 2
}

// This class holds data that is kept by the system to prioritize Surface baking.
class SurfaceEntry
{
    public __GameObject__  m_Surface; // the __GameObject__ corresponding to this Surface
    public int         m_Id; // ID for this Surface
    public DateTime    m_UpdateTime; // update time as reported by the system
    public BakedState  m_BakedState;
    public const float c_Extents = 5.0f;
}

public class SMSample : MonoBehaviour
{
    // This observer is the window into the Spatial Mapping world.
    SurfaceObserver m_Observer;

    // This dictionary contains the set of known Spatial Mapping Surfaces.
    // Surfaces are updated, added, and removed by the system on a regular basis.
    Dictionary<int, SurfaceEntry> m_Surfaces;

    // This is the material with which the system draws baked Surfaces.
    public Material m_drawMat;

    // This flag is used by the Spatial Mapping system to postpone requests if a bake is in pro
    // Baking mesh data can take multiple frames.  This sample prioritizes baking request
    // order based on Surface data Surfaces and only issues a new request
    // if there are currently no requests being processed by the system.
    bool m_WaitingForBake;
```

```
    // This is the last time the SurfaceObserver was updated by the system. It updates no
    // more than every two seconds
    float m_lastUpdateTime;

    void Start ()
    {
        m_Observer = new SurfaceObserver ();
        m_Observer.SetVolumeAsAxisAlignedBox (new Vector3(0.0f, 0.0f, 0.0f),
            new Vector3 (SurfaceEntry.c_Extents, SurfaceEntry.c_Extents, SurfaceEntry.c_Extents
        m_Surfaces = new Dictionary<int, SurfaceEntry> ();
        m_WaitingForBake = false;
        m_lastUpdateTime = 0.0f;
    }

    void Update ()
    {
        // Avoid calling Update on a SurfaceObserver too frequently.
        if (m_lastUpdateTime + 2.0f < Time.realtimeSinceStartup)
        {
            // This block makes the observation volume follow the __camera__.
            Vector3 extents;
            extents.x = SurfaceEntry.c_Extents;
            extents.y = SurfaceEntry.c_Extents;
            extents.z = SurfaceEntry.c_Extents;
            m_Observer.SetVolumeAsAxisAlignedBox (Camera.main.transform.position, extents);

            try
            {
                m_Observer.Update (SurfaceChangedHandler);
            }
            catch
            {
                // Update can throw an exception if the specified callback is bad.
                Debug.Log ("Observer update failed unexpectedly!");
            }

            m_lastUpdateTime = Time.realtimeSinceStartup;
        }
        if (!m_WaitingForBake)
        {
            // Prioritize older adds over other adds over updates.
            SurfaceEntry bestSurface = null;
            foreach (KeyValuePair<int, SurfaceEntry> surface in m_Surfaces)
            {
                if (surface.Value.m_BakedState != BakedState.Baked)
                {
                    if (bestSurface == null)
                    {
                        bestSurface = surface.Value;
                    }
                    else
                    {
                        if (surface.Value.m_BakedState < bestSurface.m_BakedState)
                        {
```

```
                        bestSurface = surface.Value;
                    }
                    else if (surface.Value.m_UpdateTime < bestSurface.m_UpdateTime)
                    {
                        bestSurface = surface.Value;
                    }
                }
            }
        }
        if (bestSurface != null)
        {
            // Fill out and dispatch the request.
            SurfaceData sd;
            sd.id.handle = bestSurface.m_Id;
            sd.outputMesh = bestSurface.m_Surface.GetComponent<MeshFilter> ();
            sd.outputAnchor = bestSurface.m_Surface.GetComponent<WorldAnchor> ();
            sd.outputCollider = bestSurface.m_Surface.GetComponent<MeshCollider> ();
            sd.trianglesPerCubicMeter = 300.0f;
            sd.bakeCollider = true;
            try
            {
                if (m_Observer.RequestMeshAsync(sd, SurfaceDataReadyHandler))
                {
                    m_WaitingForBake = true;
                }
                else
                {
                    // A return value of false when requesting meshes
                    // typically indicates that the specified
                    // Surface ID is invalid.
                    Debug.Log(System.String.Format ("Bake request for {0} failed.  Is {0} a
                }
            }
            catch
            {
                // Requests can fail you do not fill out the data struct properly
                Debug.Log (System.String.Format("Bake for id {0} failed unexpectedly!", bes
            }
        }
    }
}

// This handler receives events when surfaces change, and propagates those events
// using the SurfaceObserver’s Update method
void SurfaceChangedHandler (SurfaceId id, SurfaceChange changeType, Bounds bounds, DateTime
{
    SurfaceEntry entry;
    switch (changeType)
    {
        case SurfaceChange.Added:
        case SurfaceChange.Updated:
        if (m_Surfaces.TryGetValue(id.handle, out entry))
        {
            // If the system as already baked this Surface, mark it as needing to be baked
```

```
            // in addition to the update time so the "next Surface to bake"
            // logic orders it correctly.
            if (entry.m_BakedState == BakedState.Baked)
            {
                entry.m_BakedState = BakedState.UpdatePostBake;
                entry.m_UpdateTime = updateTime;
            }
        }
        else
        {
            // This is a brand new Surface so create an entry for it.
            entry = new SurfaceEntry ();
            entry.m_BakedState = BakedState.NeverBaked;
            entry.m_UpdateTime = updateTime;
            entry.m_Id = id.handle;
            entry.m_Surface = new GameObject (System.String.Format("Surface-{0}", id.handle
            entry.m_Surface.AddComponent<MeshFilter> ();
            entry.m_Surface.AddComponent<MeshCollider> ();
            MeshRenderer mr = entry.m_Surface.AddComponent<MeshRenderer> ();
            mr.shadowCastingMode = ShadowCastingMode.Off;
            mr.receiveShadows = false;
            entry.m_Surface.AddComponent<WorldAnchor> ();
            entry.m_Surface.GetComponent<MeshRenderer> ().sharedMaterial = m_drawMat;
            m_Surfaces[id.handle] = entry;
        }
        break;

        case SurfaceChange.Removed:
        if (m_Surfaces.TryGetValue(id.handle, out entry))
        {
            m_Surfaces.Remove (id.handle);
            Mesh mesh = entry.m_Surface.GetComponent<MeshFilter> ().mesh;
            if (mesh)
            {
                Destroy (mesh);
            }
            Destroy (entry.m_Surface);
        }
        break;
    }
}

void SurfaceDataReadyHandler(SurfaceData sd, bool outputWritten, float elapsedBakeTimeSecon
{
    m_WaitingForBake = false;
    SurfaceEntry entry;
    if (m_Surfaces.TryGetValue(sd.id.handle, out entry))
    {
        // These two asserts check that the returned filter and WorldAnchor
        // are the same as those used by the system to request the data. This should always
        // be true unless you have changed code to replace or destroy them.
        Assert.IsTrue (sd.outputMesh == entry.m_Surface.GetComponent<MeshFilter>());
        Assert.IsTrue (sd.outputAnchor == entry.m_Surface.GetComponent<WorldAnchor>());
        entry.m_BakedState = BakedState.Baked;
```

```
        }
        else
        {
            Debug.Log (System.String.Format("Paranoia:  Couldn't find surface {0} after a bake
            Assert.IsTrue (false);
        }
    }
}
```

Note: Calling the SurfaceObserver's Update method can be resource-intensive, so you should try not to do it more than an application requires. Calling this method once every three seconds should be enough for most applications.

2018–05–01 Page published with editorial review

Spatial Mapping for Hololens documentation updated in 2017.3

Spatial Mapping common troubleshooting issues

Leave feedback

Spatial Mapping generates a large amount of data. This has an impact on your application's speed and performance. This section discusses common issues that can arise from using Spatial Mapping, and guidance on how to avoid those issues.

Collision data generation takes a long time

Generating **collision** data requires more CPU processing time than any other aspect of Spatial Mapping. Ensuring that **collision** data is only requested when necessary optimizes usage of CPU resources and increase battery life.

Best practice solution:

Only request collision data where necessary. Avoiding collision data requests when they are not necessary decreases the latency for other Spatial Mapping requests, and prolongs battery life.

Use a **Surface's** update time and bounding boxes to prioritize data requests, and only request **Surfaces** that are high priority.

High triangle densities generate large amounts of geometry

High values of trianglesPerCubicMeter, set through the SurfaceData struct when requesting **Surface** data using the RequestMeshAsync method, generate a very large amount of geometry in your **Scene**. This is especially true in spaces containing many objects (like a cluttered office). Large amounts of geometry in your Scene increase data generation latency and the memory usage of your application. Higher **Mesh** densities in your Scene can also slow down run-time systems such as **rendering** and physics, which can affect performance.

Best practice solution:

Use the minimum resolution of Spatial Mapping data that your application requires. To do this, you need to test your application and the accuracy of the generated Spatial Mapping mesh, but the result is a balance of accuracy and performance that ultimately offers a much more positive user experience. Lower resolutions of generated Meshes result in less CPU time being spent by your application when the Mesh updates. This reduces the power consumption of your device, increases battery life, and also reduces latency in retrieving Mesh data. Lower resolution Meshes also use less memory and have a less impact on run-time systems such as rendering and physics (which expects low complexity geometry).

Queuing too many Mesh requests results in unnecessary work

SurfaceObservers report all added, updated, and removed **Surfaces** within their volume when you call the SurfaceObserver.Update method.

This adds a list of all changed **Surfaces** to the work queue and can result in unused **Surfaces** remaining in the work queue after Spatial Mapping has removed them. These **Surfaces** still take up CPU time when moving through the system, but do not generate any Mesh data. This increases the latency of any waiting requests.

Best practice solution:

Limit the number of **Surfaces** in the work queue where possible. Querying Meshes is an expensive operation with high latency, so keeping only a single RequestMeshAsync request in use at any given time minimizes any slowdown that these operations cause. Applications can use a **Surface's** reported update time and bounds to prioritize `RequestMeshAsync` calls.

It is also important to prioritize **Surface** data requests so that your application receives the most important data first. Examples of common ways to prioritize include:

Prioritizing new **Surfaces** above updating existing ones.

Prioritizing **Surfaces** that are closer to the user over those further away.

Overlapping SurfaceObserver volumes create duplicate RequestMeshAsync calls

SurfaceObservers report changes for all **Surfaces** that overlap their volume. A **Surface** can overlap multiple `SurfaceObserver` volumes if they are close together. This makes it possible for application code to request the same **Surfaces** multiple times, which can lead to performance issues.

Best practice solution:

Use a single work submission queue for requests from `SurfaceObserver`.

For many applications, a single `SurfaceObserver` is often all you need.

Using a single `SurfaceObserver` can help you reduce the complexity of developing your application. However, there are several advanced uses for **Spatial Mapping** that require multiple `SurfaceObserver` members. In these cases, you should use a single work queue for all `SurfaceObserver` members in your Scene in order to uniquely prioritize Mesh requests.

Updating a SurfaceObserver generates no onSurfaceChanged callbacks

This is a common issue, typically caused by issues in the set-up process.

Best practice solution:

Set a valid volume on your `SurfaceObserver`, using a method such as SetVolumeAsAxisAlignedBox.

You should also make sure that you enable the **Spatial Perception** capability in **Publishing Settings** (menu: **File > Build Settings > Player Settings > Publishing Settings)**.

For more information relating to Capability and Publishing Settings for WMR, see Unity's Windows Mixed Reality quick starter guide.

2018–05–01 Page published with editorial review

Spatial Mapping for Hololens documentation updated in 2017.3

Unity XR input

Leave feedback

This section provides information on all Unity supported input devices used to interact in **Virtual Reality**, Augmented Reality and Mixed Reality applications.

Input for Windows Mixed Reality

Leave feedback

Windows Mixed Reality motion controllers allow users to interact in Mixed Reality applications, allowing precise, low latency tracking of movement within the field of view (FOV) of their **Windows Mixed Reality** headset. This is achieved using the sensors built into the headset.

To allow native Unity Editor support for Windows Mixed Reality input hardware, the Unity VR subsystem presents three inputs as separate joysticks. Use the UnityEngine.Input class to read the axis and button values of these inputs.

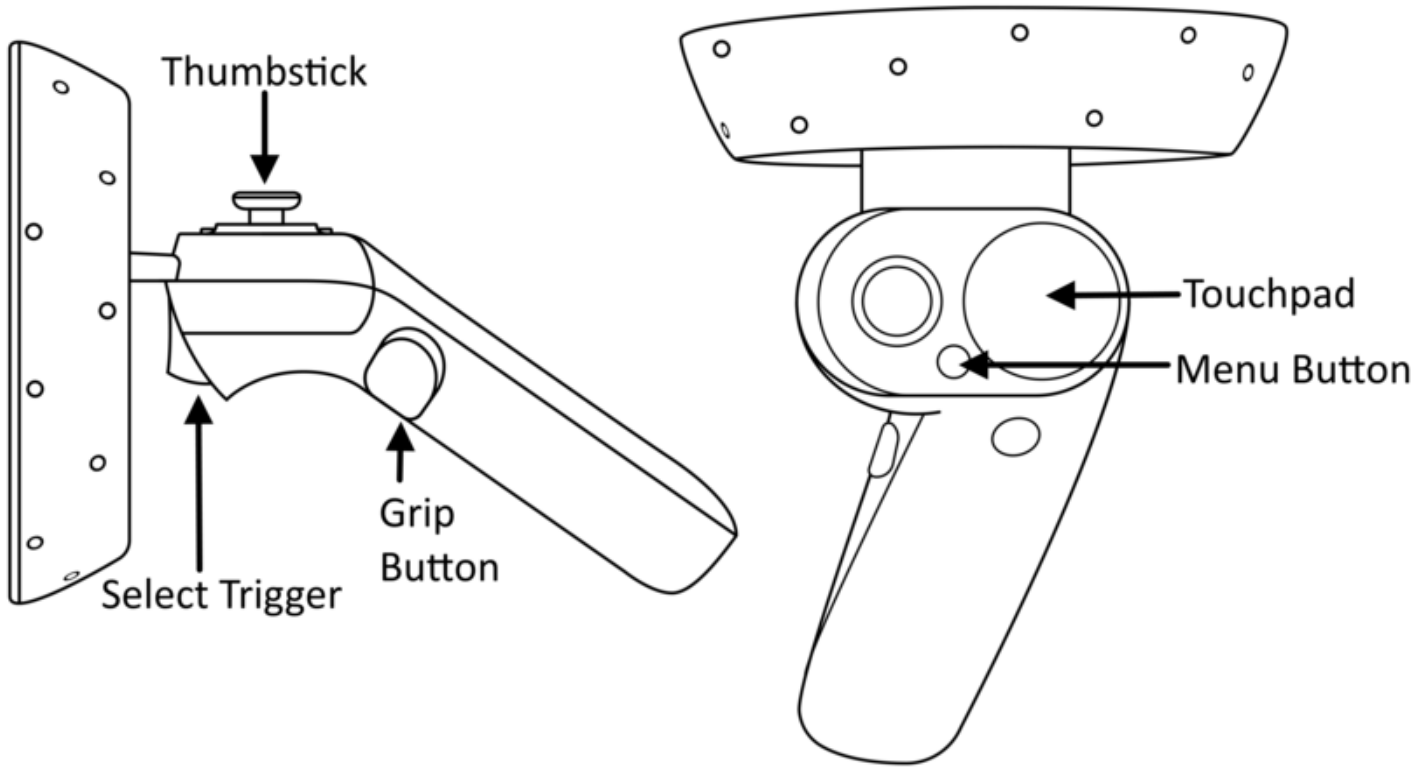
Controller naming conventions and detection

When properly configured and connected to your computer, the motion controllers appear in the list returned by UnityEngine.Input.GetJoystickNames() as **Spatial Controller - Left** and **Spatial Controller - Right**. For information on correctly configuring and connecting the motion controllers, see the Windows Developer Center documentation.

In Unity you can check the availability of the controllers through script by periodically checking for their presence in the list of joystick names. When the controllers are turned off, or their batteries are removed, an empty string replaces their name in the list returned by UnityEngine.Input.GetJoystickNames(). When the controllers are turned back on, their name re-appears in the list.

Unity Input system mappings

The image below shows the buttons available on your Windows Mixed Reality controller.



Windows Mixed Reality Controller buttons (image courtesy of developer.microsoft.com)

The table below shows the interaction types, button IDs, axis and value ranges for each input provided by the controller in Unity.

Hardware Feature	Interaction Type	Unity Button ID	Unity Axis ID	Unity Axis Value Range
Touchpad	Touch	Left: 18 Right: 19	-	-
Touchpad	Press	Left: 16 Right: 17	-	-
Touchpad	Horizontal Movement	-	Left: 17 Right: 19	–1.0 to 1.0
Touchpad	Vertical Movement	-	Left: 18 Right: 20	–1.0 to 1.0
Thumbstick	Press	Left: 8 Right: 9	-	-
Thumbstick	Horizontal Movement	-	Left: 1 Right: 4	–1.0 to 1.0
Thumbstick	Vertical Movement	-	Left: 2 Right: 5	–1.0 to 1.0
Select Trigger	Press	Left: 14 Right: 15	-	-
Select Trigger	Squeeze	-	Left: 9 Right: 10	0.0 to 1.0
Grip button	Press	Left: 4 Right: 5	-	-

Hardware Feature	Interaction Type	Unity Button ID	Unity Axis ID	Unity Axis Value Range
Grip button	Squeeze	-	Left: 11 Right: 12	0.0 and 1.0*
Menu button	Press	Left: 6 Right: 7	-	-

Windows Mixed Reality controller input details for Unity

*The Grip squeeze axis is a binary control, so it only reports values of 0 or 1, with no values in between.

The table below lists the different axes available when using Window Mixed Reality controller inputs, along with the positive and negative directions for each axis.

Axis	Positive Direction	Negative Direction
Horizontal	Left	Right
Vertical	Up	Down

For more information on using the Windows MR motion controllers with Unity, visit the Windows Developer Center documentation on Motion Controllers in Unity.

2017–11–21 Page published with editorial review

New functionality in 2017.2

Input for OpenVR controllers

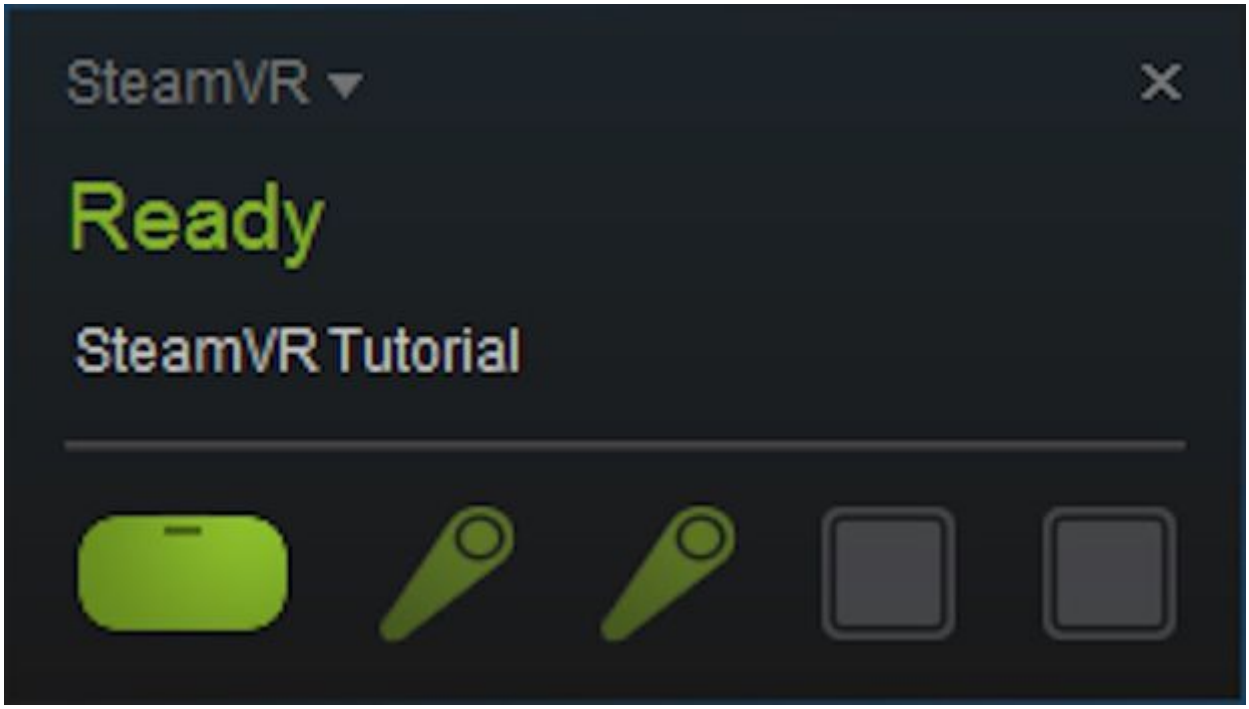
Leave feedback

For the Unity Editor to support OpenVR tracked controllers, the Unity VR subsystem presents VR controller inputs as separate joysticks. Use the UnityEngine.Input class to access the axis and button values.

OpenVR's Unity integration doesn't refer to any specific hardware when presenting axis and button states. This page provides the axis and button mappings for the three types of controllers supported by OpenVR: HTC Vive, **Oculus** Touch, and Valve Knuckles Controllers.

Naming convention and detection

When properly configured and connected, any OpenVR-compatible controllers are internally named as either **OpenVR Controller - Left** or **OpenVR Controller - Right**. Access this name through the list returned by UnityEngine.Input.GetJoystickNames(). When available, these controllers appear highlighted in green in the SteamVR status menu when tested with Steam. To access this menu you must have both Steam and SteamVR installed and running on your machine.



Steam VR status menu

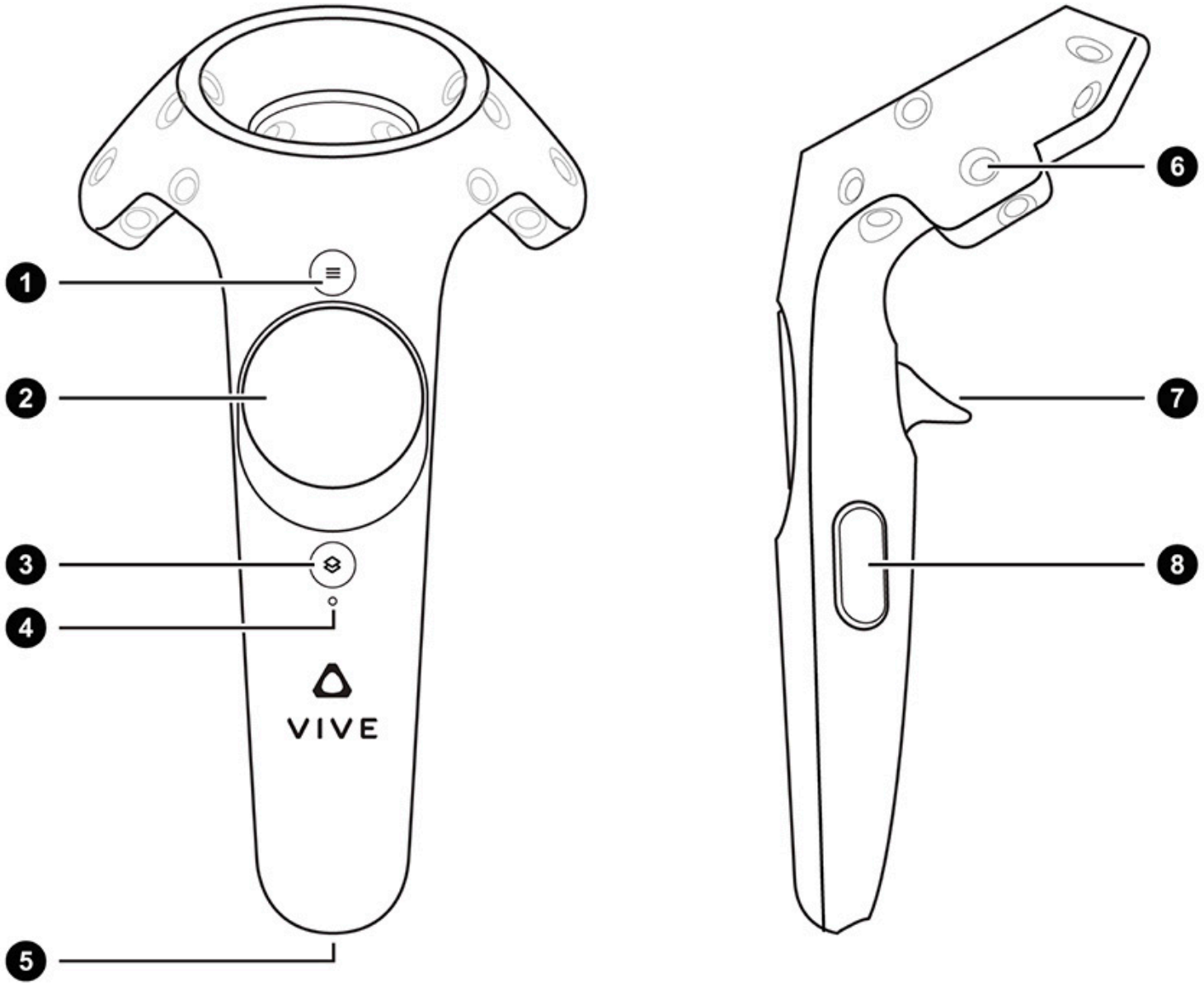
You can test the availability of these controllers by periodically checking for their presence in the list of joystick names through script. When the controllers turn off, or when you remove their batteries, an empty string replaces their name in the list returned by UnityEngine.Input.GetJoystickNames(). When the controllers turn on again, their name reappears in the list of returned joysticks.

Unity input system mappings

This section provides diagrams for each type of controller supported by OpenVR devices, along with information on the internal Unity input mapping for each controller button.

HTC Vive controllers

The diagram below displays the different inputs available on HTC Vive controllers for use in VR applications.

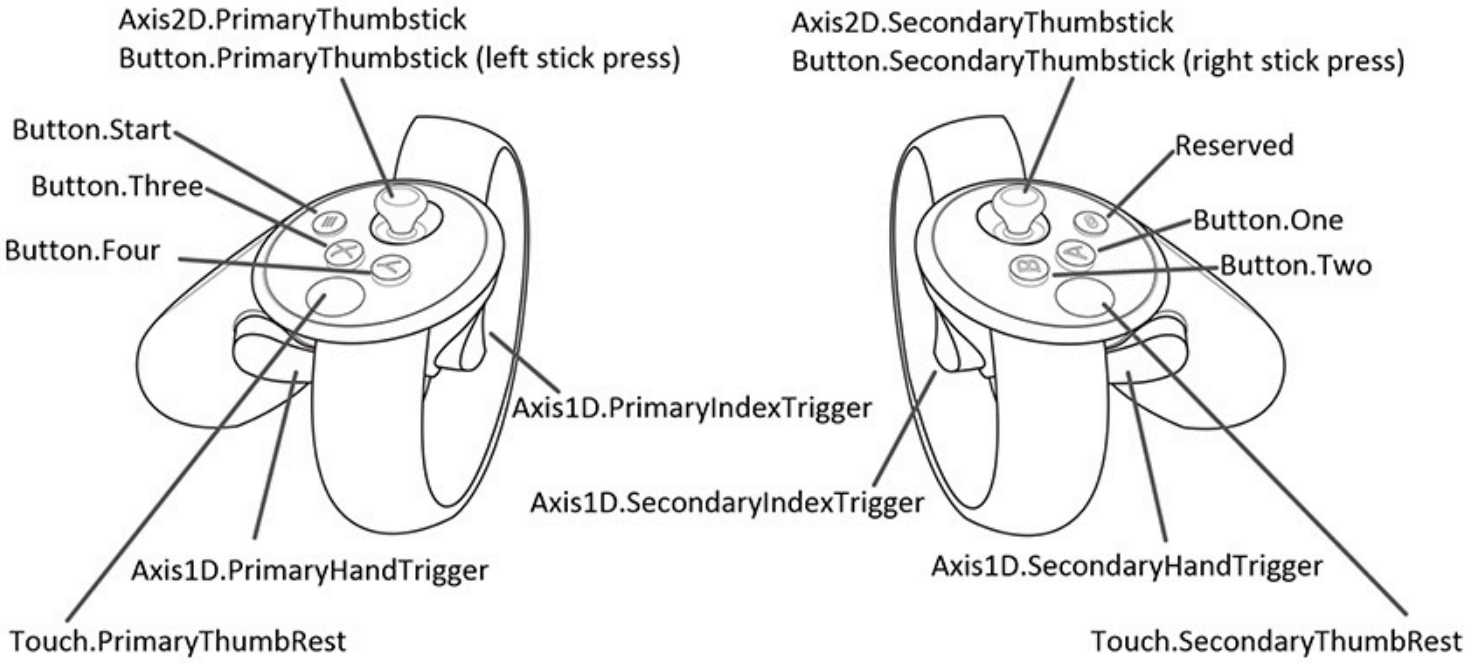


HTC Vive controller inputs mapping (Image courtesy of developer.viveport.com)

1 Menu Button
2 Trackpad
3 System button
4 Status light
5 Micro-USB port
6 Tracking sensor
7 Trigger
8 Grip button

Oculus Touch controllers

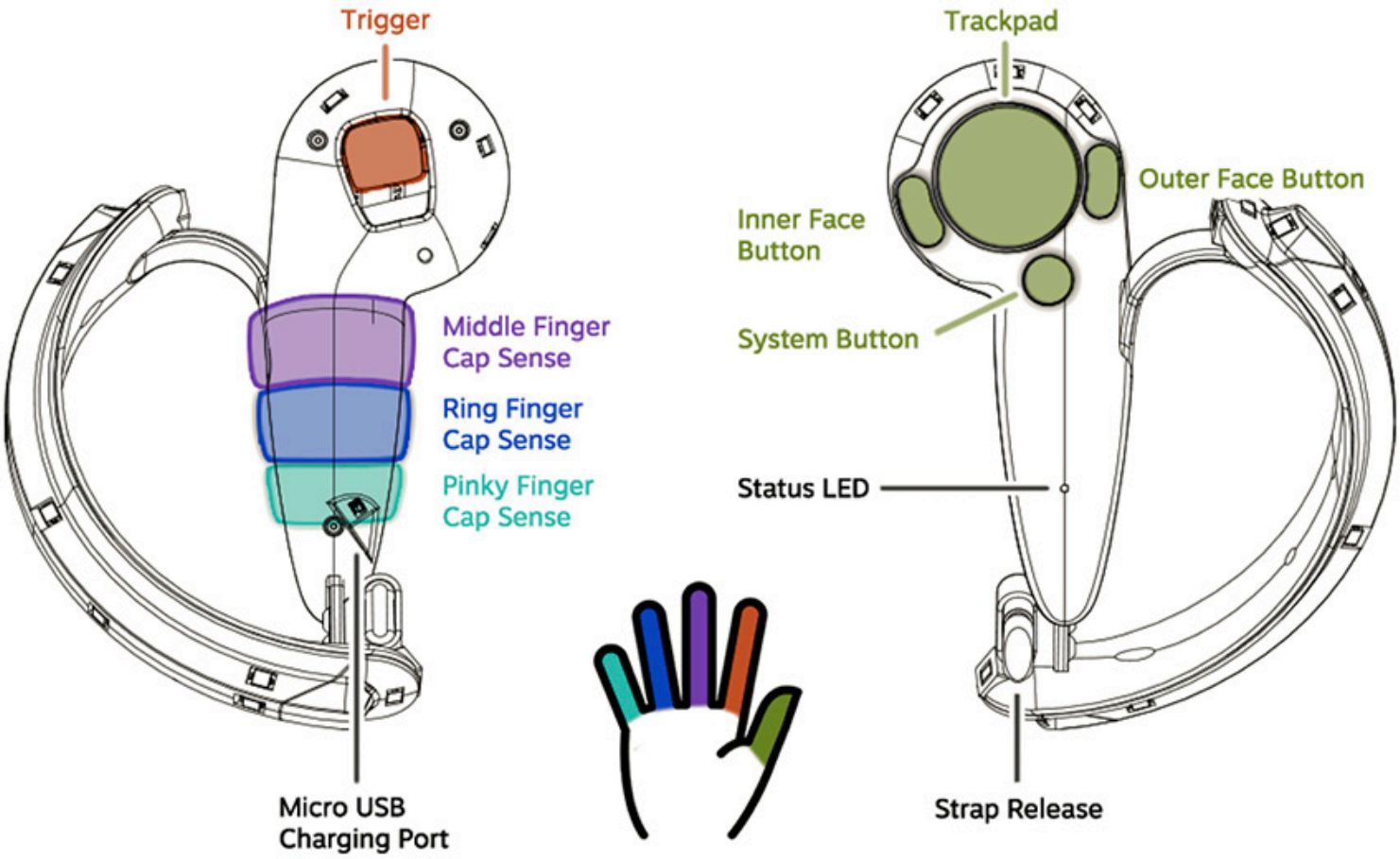
The diagram below displays the different inputs available on Oculus Touch controllers for use in VR applications.



Oculus Touch controller input mapping (Image courtesy of developer.oculus.com)

Knuckles controllers

The diagram below displays the different inputs available when using Knuckles controllers in VR applications.



Valve Knuckles Controller input mapping (Image courtesy of steamcommunity.com)

The table below lists controller inputs for each OpenVR-supported controller, their interaction types, Unity axis and button IDs, along with the value range for each axis.

HTC Vive Controller	Oculus Touch Controller	Valve Knuckles Controller	Interaction Type	Unity Button ID	Unity Axis ID	Unity Axis Value Range
Left Controller Menu Button (1)	Button.Three	Left Controller Inner Face Button	Press	2		
Right Controller Menu Button (1)	Button.One	Right Controller Inner Face Button	Press	0		
		Left Controller Outer Face Button	Press	3		
		Right Controller Outer Face Button	Press	1		
Left Controller Trackpad (2)	Button.PrimaryThumbstick	Left Controller Trackpad	Press	8		
Right Controller Trackpad (2)	Button.SecondaryThumbstick	Right Controller Trackpad	Press	9		
Left Controller Trackpad (2)	Button.PrimaryThumbstick	Left Controller Trackpad	Touch	16		
Right Controller Trackpad (2)	Button.SecondaryThumbstick	Right Controller Trackpad	Touch	17		
Left Controller Trackpad (2)	Axis2D.PrimaryThumbstick	Left Controller Trackpad	Horizontal Movement		1	–1.0 to 1.0
Left Controller Trackpad (2)	Axis2D.PrimaryThumbstick	Left Controller Trackpad	Vertical Movement		2	–1.0 to 1.0
Right Controller Trackpad (2)	Axis2D.SecondaryThumbstick	Right Controller Trackpad	Horizontal Movement		4	–1.0 to 1.0
Right Controller Trackpad (2)	Axis2D.SecondaryThumbstick	Right Controller Trackpad	Vertical Movement		5	–1.0 to 1.0
Left Controller Trigger (7)	Axis1D.PrimaryIndexTrigger	Left Controller Trigger	Touch	14		
Right Controller Trigger (7)	Axis1D.SecondaryIndexTrigger	Right Controller Trigger	Touch	15		
Left Controller Trigger (7)	Axis1D.PrimaryIndexTrigger	Left Controller Trigger	Squeeze		9	0.0 to 1.0
Right Controller Trigger (7)	Axis1D.SecondaryIndexTrigger	Right Controller Trigger	Squeeze		10	0.0 to 1.0
Left Controller Grip Button (8)	Axis1D.PrimaryHandTrigger	Left Controller Grip Average	Squeeze		11	0.0 to 1.0

HTC Vive Controller	Oculus Touch Controller	Valve Knuckles Controller	Interaction Type	Unity Button ID	Unity Axis ID	Unity Axis Value Range
Right Controller Grip Button (8)	Axis1D.SecondaryHandTrigger	Right Controller Grip Average	Squeeze		12	0.0 to 1.0
		Left Controller Index Finger Cap Sensor			20	0.0 to 1.0
		Right Controller Index Finger Cap Sensor			21	0.0 to 1.0
		Left Controller Middle Finger Cap Sensor			22	0.0 to 1.0
		Right Controller Middle Finger Cap Sensor			23	0.0 to 1.0
		Left Controller Ring Finger Cap Sensor			24	0.0 to 1.0
		Right Controller Ring Finger Cap Sensor			25	0.0 to 1.0
		Left Controller Pinky Finger Cap Sensor			26	0.0 to 1.0
		Right Controller Pinky Finger Cap Sensor			27	0.0 to 1.0

Note: Any hardware features not listed in the table above are not exposed through the OpenVR API and are therefore not exposed through Unity's input system.

Important controller differences

When developing applications to support OpenVR devices, it is important to note certain differences in input event triggers and response rates for buttons on all three controllers.

Unity Touch Input events differ for each platform controller:

The Event System generates Input Touch events for the triggers on HTC Vive controllers when the user begins squeezing the trigger.
The Event System generates Input Touch events for the triggers on Oculus Touch and Valve Knuckles controllers when Unity Input detects a touch - no trigger squeeze is necessary.
The grip trigger differs between platforms:

The HTC Vive controllers each have two grip buttons but both map to the same axis. The value for these buttons is 0.0 when not pressed and 1.0 when pressed (intermediate values between 0.0 and 1.0 are not possible).

The grip on the Oculus Touch Controller is an analog trigger with a range from 0.0 to 1.0 (intermediate values are possible).

The grip on the Valve Knuckles Controller is a weighted average of the input values for each of the individual finger touch sensors.

Input for Oculus

Leave feedback

Oculus Rift has three inputs: two **Oculus** Touch Controllers, and one **Oculus** Remote. To allow native Unity Editor support for **Oculus** Rift input hardware, the Unity VR subsystem presents these three inputs as separate joysticks: Use the UnityEngine.Input class to read the axis and button values of these inputs.

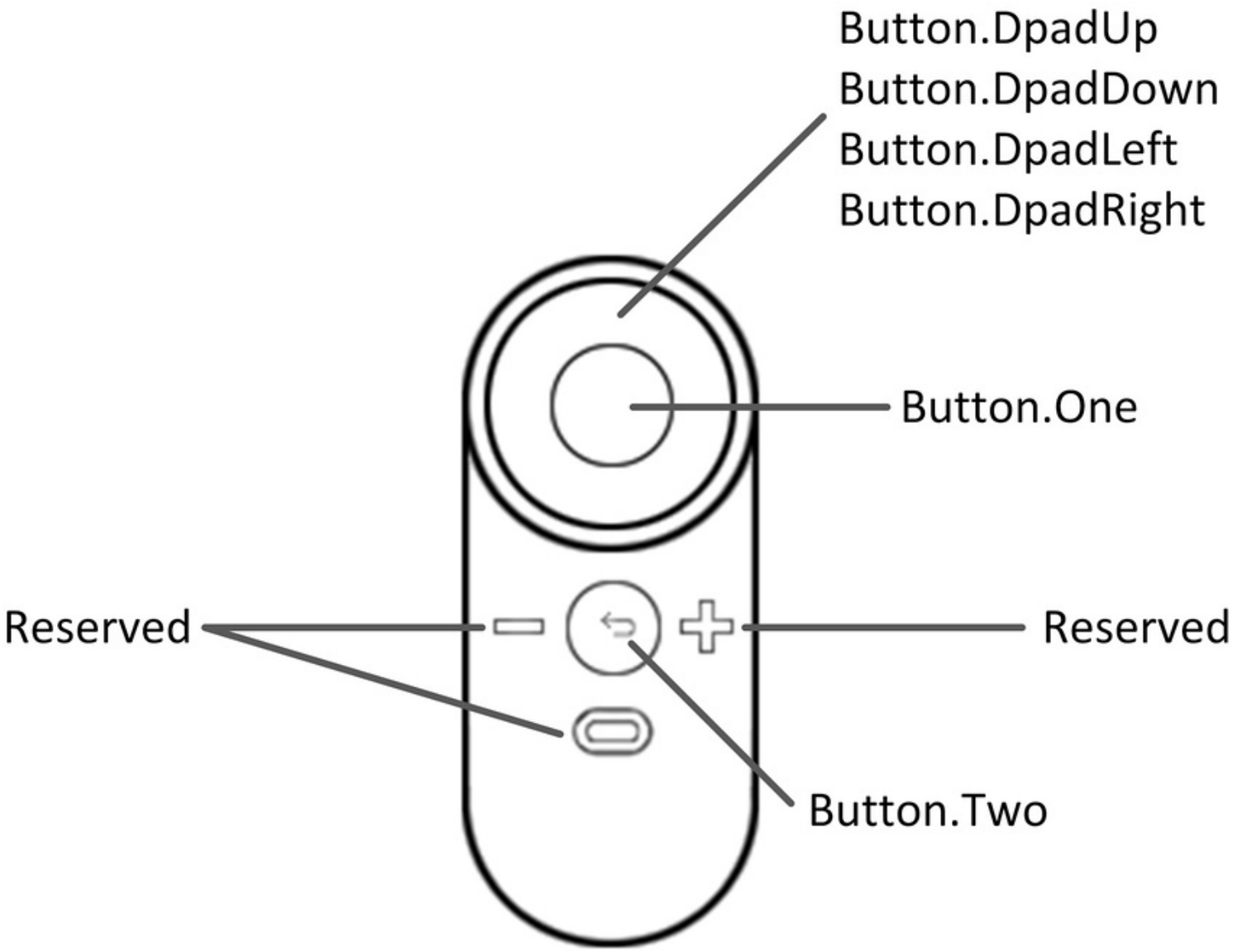
Naming convention and detection

When properly configured and connected with your operating system, the Oculus Touch Controllers appear in the list returned by UnityEngine.Input.GetJoystickNames() as Oculus Touch - Left and Oculus Touch - Right, and the Oculus Remote appears as Oculus Remote. For information on correctly configuring and connecting the Touch Controllers and Remote, see the Oculus Developer Center documentation on Pairing the Oculus Touch Controllers and the Oculus Support Center guidance on the Oculus Remote.

Unity script code can test for the availability of the Touch Controllers by periodically checking for their presence in the list of joystick names. When the Touch Controllers are turned off, or their batteries are removed, an empty string replaces their name in the list returned by UnityEngine.Input.GetJoystickNames(). When the Touch Controllers are turned on again, their name re-appears in the list.

Unity input system mappings

Oculus Remote

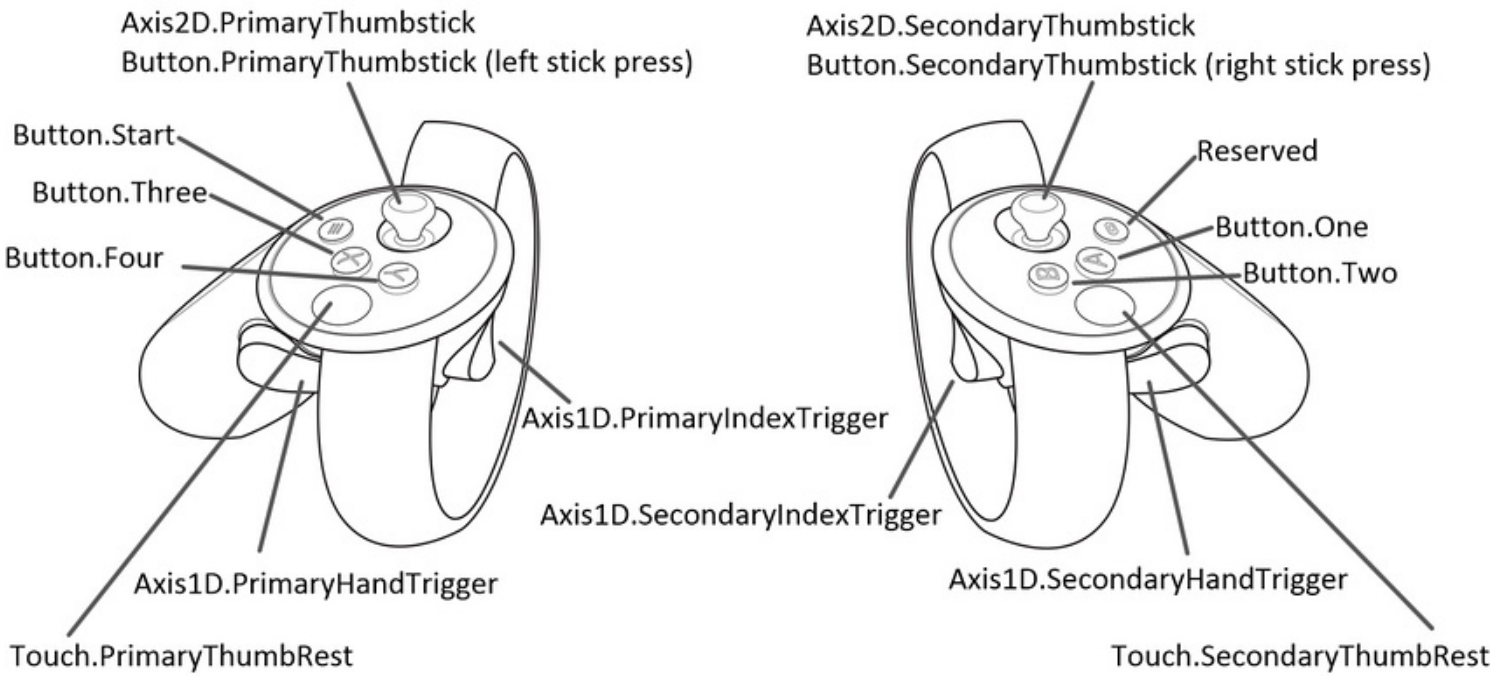


(Image courtesy of developer.oculus.com)

Hardware Feature	Unity Button ID	Unity Axis ID	Unity Axis Value when Pressed	Xbox Controller Analogue
Button.DpadUp	-	6	1.0	D-Pad Up
Button.DpadDown	-	6	–1.0	D-Pad Down
Button.DpadLeft	-	5	–1.0	D-Pad Left
Button.DpadRight	-	5	1.0	D-Pad Right
Button.One	0	-	-	A Button
Button.Two	1	-	-	B Button

Oculus Touch Controllers

Note: The two Touch Controllers have a similar set of controls to the Xbox controller, so Unity's Oculus Touch Controller mapping closely imitates those.



(Image courtesy of developer.oculus.com)

Hardware Feature	Interaction Type	Unity Button ID	Unity Axis ID	Unity Axis Range	Xbox Controller Analog
Button.One	Press	0	-	-	A Button
Button.One	Touch	10	-	-	-
Button.Two	Press	1	-	-	B Button
Button.Two	Touch	11	-	-	-
Button.Three	Press	2	-	-	X Button
Button.Three	Touch	12	-	-	-
Button.Four	Press	3	-	-	Y Button
Button.Four	Touch	13	-	-	-
Button.Start	Press	7	-	-	Start Button
Button.PrimaryThumbstick	Press	8	-	-	Left Stick Press
Button.PrimaryThumbstick	Touch	16	-	-	-
Button.PrimaryThumbstick	Near Touch	-	15	1.0 When Near Touch Is Detected; 0.0 Otherwise	-
Button.SecondaryThumbstick	Press	9	-	-	Right Stick Press
Button.SecondaryThumbstick	Touch	17	-	-	-
Button.SecondaryThumbstick	Near Touch	-	16	1.0 When Near Touch Is Detected; 0.0 Otherwise	-
Touch.PrimaryThumbRest	Touch	18	-	-	-
Touch.PrimaryThumbRest	Near Touch	-	15	1.0 When Near Touch Is Detected; 0.0 Otherwise	-
Touch.SecondaryThumbRest	Touch	19	-	-	-

Hardware Feature	Interaction Type	Unity Button ID	Unity Axis ID	Unity Axis Range	Xbox Controller Analog
Touch.SecondaryThumbRest	Near Touch	-	16	1.0 When Near Touch Is Detected; 0.0 Otherwise	-
Axis1D.PrimaryIndexTrigger	Touch	14	-	-	-
Axis1D.PrimaryIndexTrigger	Near Touch	-	13	1.0 When Near Touch Is Detected; 0.0 Otherwise	-
Axis1D.PrimaryIndexTrigger	Squeeze	-	9	0.0 to 1.0	Squeeze Left Trigger
Axis1D.SecondaryIndexTrigger	Touch	15	-	-	-
Axis1D.SecondaryIndexTrigger	Near Touch	-	14	1.0 When Near Touch Is Detected; 0.0 Otherwise	-
Axis1D.SecondaryIndexTrigger	Squeeze	-	10	0.0 to 1.0	Squeeze Right Trigger
Axis1D.PrimaryHandTrigger	Squeeze	-	11	0.0 to 1.0	-
Axis1D.SecondaryHandTrigger	Squeeze	-	12	0.0 to 1.0	-
Axis2D.PrimaryThumbstick	Horizontal Movement	-	1	–1.0 to 1.0	Move Left Stick
Axis2D.PrimaryThumbstick	Vertical Movement	-	2	–1.0 to 1.0	Move Left Stick
Axis2D.SecondaryThumbstick	Horizontal Movement	-	4	–1.0 to 1.0	Move Right Stick
Axis2D.SecondaryThumbstick	Vertical Movement	-	5	–1.0 to 1.0	Move Right Stick

XR API reference

Leave feedback

XR.XRSettings
XR.XRDevice
XR.InputTracking

Mixed Reality Devices

Leave feedback

Unity provides built-in support for a number of Mixed Reality (MR) devices.

Device availability is on a per-platform basis, meaning that not every device is available for every platform. Multiple devices are available on certain platforms, and you can choose which MR devices your app will support from the available list.

At runtime, only one device can be active at any given time. However, it is possible to switch between devices if you have chosen to support multiple devices in your application.

MR device information

XRDevice.family is a string corresponding to the current XRSettings.loadedDeviceName. A family can have multiple models, which have different characteristics. The model can be accessed with XRDevice.model.

VR overview

Leave feedback

Unity VR lets you target **virtual reality** devices directly from Unity, without any external plug-ins in projects. It provides a base API and feature set with compatibility for multiple devices. It has been designed to provide forward compatibility for future devices and software.

Note: XR is an umbrella term which encompassess virtual reality (VR), augmented reality (AR), and Mixed Reality (MR). We are in the process of updating our XR documentation, but for now this documentation refers mainly to just VR.

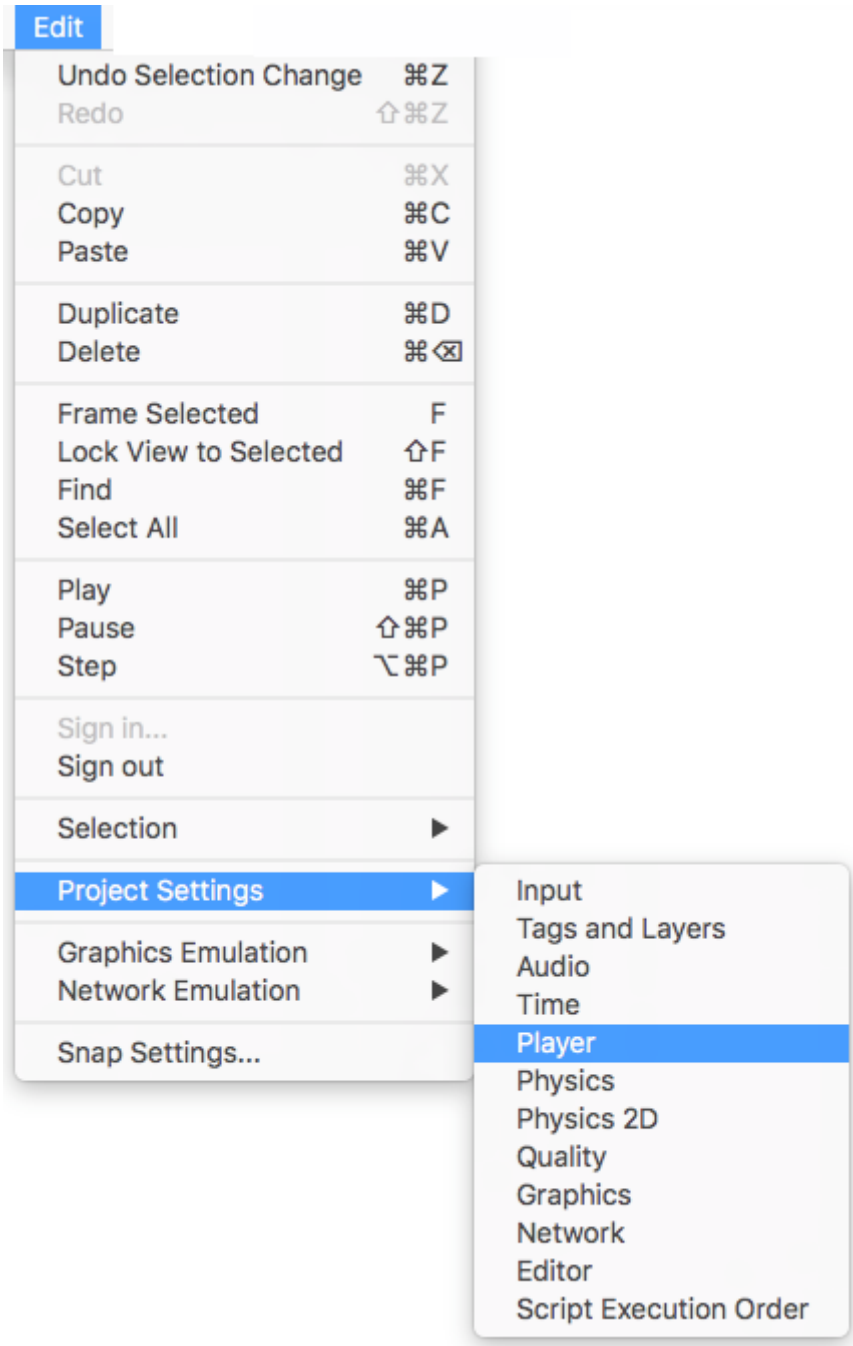
Unity's XR API has been updated to reflect the broader umbrella term "XR" but much of the documentation currently still uses the term "VR". The XR API surface is minimal by design, but will expand as XR continues to grow.

By using the native VR support in Unity, you gain:

Stable versions of each VR device
A single API interface to interact with different VR devices
A clean project folder with no external **plugin** for each device
The ability to include and switch between multiple devices in your applications
Increased performance (Lower-level Unity engine optimizations are possible for native devices) ot

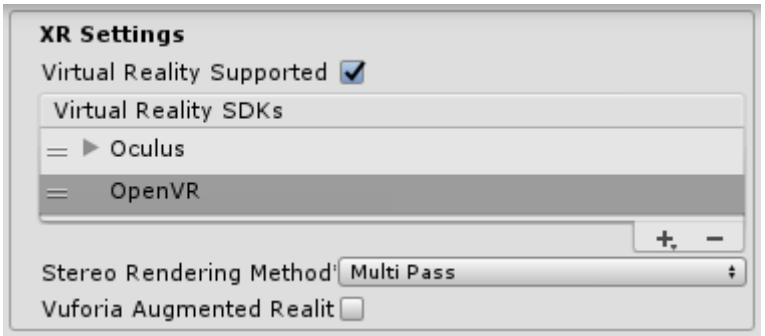
Enabling Unity VR support

To enable native VR support for your game builds and the Editor, open the Player Settings (menu: **Edit** > **Project Settings** > **Player**). Select **XR Settings** and check the **Virtual Reality Supported** checkbox. Set this for each build target. Enabling virtual reality support in a standalone build does not also enable the support for Android (and vice-versa).



Navigating to Player Settings from the Unity Editor

Use the **Virtual Reality SDKs** list displayed below the checkbox to add and remove VR devices for each build target. The order of the list is the order that Unity tries to enable VR devices at runtime. The first device that initializes properly is the one enabled. This list order is the same in the built player.



Enabling Virtual Reality Support through XR Settings

Built applications: Choosing startup device

Your built application initializes and enables devices in the same order as the **Virtual Reality SDKs** list in **XR Settings** (see Enabling VR Support, above). Devices not present in the list at build time are not available in the final build. Device **None** is equivalent to a non-VR application (that is, a normal Unity application) and can be switched to during runtime without including it in the list; the exception to this is **None**.

Including **None** as a device in the list allows you to default to a non-VR application before attempting a VR device's initialization. If you place **None** at the top of the list, the application starts with VR disabled. Then, you can then enable and disable VR devices that are present in your list through script using XR.XRSettings.LoadDeviceByName.

If initialization of the device you attempt to switch to fails, Unity disables VR with that device still set as the active VR device. When you switch devices (XRSettings.LoadDeviceByName) or enable XR (XR.XRSettings-enabled), the built application attempts to initialize again.

The command line argument used to launch a specific device is:

```
-vrmode DEVICETYPE
```

where **DEVICETYPE** is one of the names from the supported XR supported devices list.

Example: `MyGame.exe -vrmode oculus`

What happens when VR is enabled

When VR is enabled in Unity, a few things happen automatically:

Automatic rendering to a head-mounted display

All Cameras in your Scene are able to render directly to the head-mounted display (HMD). View and Projection matrices are automatically adjusted to account for head tracking, positional tracking and field of view.

It is possible to disable rendering to the HMD using the Camera component's **stereoTargetEye** property. Alternatively, you can set the Camera to render to a Render Texture using the **Target Texture** property.

Use the stereoTargetEye property to set the Camera to only render a specific eye to the HMD. This is useful for special effects such as a sniper scope or stereoscopic videos. To achieve this, add two Cameras to the Scene: one targeting the left eye, the other targeting the right eye. Set layer masks to configure what is sent to each eye.

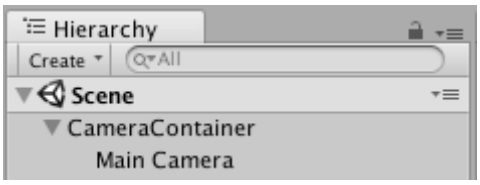
Automatic head-tracked input

Head tracking and the appropriate field of view (FOV) is automatically applied to the Camera if your device is head-mounted. You can manually set the FOV to a specific value, but you cannot set the Camera's transform values directly. See the section below on Understanding the Camera to learn more.

Head tracking and positional tracking are automatically applied, so that the position and orientation most closely matches the user's position and orientation before the frame is rendered. This gives a good VR experience, and prevents the user from experiencing nausea.

Understanding the camera

The Camera Transform is overridden with the head-tracked pose. To move or rotate the Camera, attach it as a child of another GameObject. This makes it so that all Transform changes to the parent of the Camera affect the Camera itself. This also applies to moving or rotating the camera using a script.



Childing the Main Camera to a Gameobject

Think of the Camera's position and orientation as where the user is looking in their neutral position.

There are differences between seated and room-scale VR experiences:

If your device supports a room-scale experience, the Camera's starting position is the center of the user's play space.

Using the seated experience, you can reset Cameras to the neutral position using XR.InputTracking.Recenter().

Each Camera that is rendering to the device automatically replaces the FOV of the Camera with the FOV that the user input in the software settings for each VR SDK. The user cannot change the field of view during runtime, because this behaviour is known to induce motion sickness.

Editor Mode

If your VR device supports Unity Editor mode, press Play in the Editor to test directly on your device.

The left eye is rendered to the Game View window if you have **stereoTargetEye** set to left or both. The right eye is rendered if you have **stereoTargetEye** set to right.

There is no automatic side-by-side view of the left and right eyes. To see a side-by-side view in the Game View, create two Cameras, set one to the left and one to the right eye, and set the viewport of display them side by side.

Note that there is overhead to running in the Editor, because Unity IDE needs to render each window, so you may experience lag or judder. To reduce editor rendering overhead, open the Game View and enable **Maximize on Play**.

The Unity Profiler is a helpful tool to get an idea of what your performance will be like when running outside of the Editor. However, the profiler itself also has overhead. The best way to review game performance is to create a build on your target platform and run it directly. You can see the best performance when running a non-development build, but development builds allow you to connect the Unity profiler for better performance profiling.

Hardware and software recommendations for VR development in Unity

Hardware

Achieving a frame rate similar to your target HMD is essential for a good VR experience. This must match the refresh rate of the display used in the HMD. If the frame rate drops below the HMD's refresh rate, it is particularly noticeable and often leads to nausea for the player.

The table below lists the device refresh rates for common VR headsets:

VR Device	*Refresh Rate*
Gear VR	60hz
Oculus Rift	90hz
Vive	90hz

Software

Windows: Windows 7, 8, 8.1, and Windows 10 are all compatible.

Android: Android OS Lollipop 5.1 or higher.

OS X: OSX 10.9+ with the Oculus 0.5.0.1 runtime. However, Oculus have paused development for OS X, so use Windows for native VR functionality in Unity.

Graphics card drivers: Make sure your drivers are up to date. Every device is keeping up with the newest drivers, so older drivers may not be supported.

Device runtime requirements

Each VR device requires that you have appropriate runtime installed on your machine. For example, to develop and run Oculus within Unity, you need to have the Oculus runtime (also known as Oculus Home) installed on your machine. For Vive, you need to have Steam and SteamVR installed.

Depending on what version of Unity you are using, the runtime versions for each specific device that is supported by Unity may differ. You can find runtime versions in the release notes of each major and minor Unity release.

With some version updates, previous runtime versions are no longer supported. This means that native Unity VR support does not work with earlier runtime versions, but continues to work with new runtime versions.

Unity native VR support does not read plug-ins from within your project's folder, so including earlier versions of the plug-in with native support will fail if you have VR support enabled. If you wish to use an earlier version with a release of Unity that no longer supports that version, disable **Native VR Support** (go to **XR Settings** and uncheck **Virtual Reality Supported**). You can then access the plug-in like any other 3rd party plugin. See the section above on Enabling Unity VR Support for more details.

VR devices

Leave feedback

Unity provides built-in support for a number of **virtual reality** devices.

Device availability is on a per-platform basis, meaning that not every device is available for every platform. Multiple devices are available on certain platforms, and you can choose which VR devices your app will support from the available list.

At runtime, only one device can be active at any given time. However, it is possible to switch between devices if you have chosen to support multiple devices in your application.

VR device information

XRDevice.family is a string corresponding to the current XRSettings.loadedDeviceName. A family can have multiple models, which have different characteristics. For example, **Oculus** has DK2, Gear VR, **etc**. The model can be accessed with XRDevice.model.

Oculus

Leave feedback

Getting started (Windows)

See the detailed documentation on the Oculus Developer website. In particular, follow the instructions for using Unity with Oculus, and see the recommended specs for **Oculus** development.

Getting started (Gear VR)

You do not need to install anything extra on your machine to deploy to Gear VR. Follow the instructions at the Samsung Developers' Website.

Make sure that you can deploy a Unity app to your Galaxy Note 5, S6 Edge+, S6, or S6 Edge (see Getting Started with Android Development):

Connect your Android device to your PC/Mac using a micro USB cable.
Create a new empty project (menu: **File > New Project**).
Switch your build platform to Android (menu: File > Build Settings).
Open the Player Settings (menu: **Edit > Project Settings > Player**). Select Other Settings and check the Virtual Reality Supported checkbox.
Add Oculus to the **Virtual Reality** SDK's list.
Create the folder `Plugins/Android/assets` under your project's `Assets` folder (note: This folder name is case-sensitive).
Include an Oculus signature file in your project in the `Plugins/Android/assets` folder.
Build and run. Insert the device into your headset and see the **skybox** with head tracking.

Getting started (OpenVR)

Steam is required to run OpenVR applications, so install Steam and SteamVR. Once SteamVR is working properly with your headset, add OpenVR to the list of supported SDKs. If you require additional functionality beyond Unity's built-in support, see the SteamVR asset store package.

Inputs

See documentation on Oculus Controllers for input control mapping.

OpenVR

Leave feedback

Getting Started (OpenVR)

Steam is required to run OpenVR applications, so install Steam and SteamVR. Once SteamVR is working properly with your headset, add **OpenVR** to the list of supported SDKs (see below to learn how to do this). If you require additional functionality beyond Unity's built-in support, see the SteamVR asset store package.

Supported Platforms

OpenVR is supported on the followng platforms:

Windows
macOS

macOS Specific

Unity OpenVR on a macOS requires the Metal graphics and 64bit application target, OpenGL is not supported.

OpenVR supports macOS 10.11.6 or later, but is optimized for macOS 10.13 High Sierra or later.

Inputs

See documentation on OpenVR Controllers for input control mapping.

Upgrading a project that contains the SteamVR Package

Enable virtual reality support. Open the **Player Settings** (menu: **Edit** > **Project Settings** > **Player**), select **Other Settings** and check the **Virtual Reality Supported** checkbox.
Use the **Virtual Reality SDK** list displayed below the checkbox to add OpenVR.
Enter Play Mode in the Editor to test the build.

2017–08–04 Page amended with limited editorial review

Supported platforms updated in 2017.2

Single Pass Stereo rendering (Double-Wide rendering)

Leave feedback

Single Pass Stereo rendering is a feature for PC and Playstation 4-based VR applications. It renders both left and right eye images at the same time into one packed **Render Texture** that is twice the width of a single eye texture. Unity renders the **Scene** twice using 2 draw calls for each **GameObject** that has a Renderer **component**, however, it only iterates through the **Scene** graph once when **rendering** for both the left and right eyes. During Single Pass Stereo **rendering**, both eyes share the work required by culling and shadow computation. There are also fewer graphics command state change switches, because the GPU renders each **GameObject** in a ping pong fashion (alternates rendering of objects between eyes).

Single Pass Stereo rendering allows the GPU to share culling for both eyes. The GPU only needs to iterate through all the GameObjects in the Scene once for culling purposes, and then renders the GameObjects that survived the culling process.

The comparison images below show the difference between normal VR rendering and Single Pass Stereo rendering.

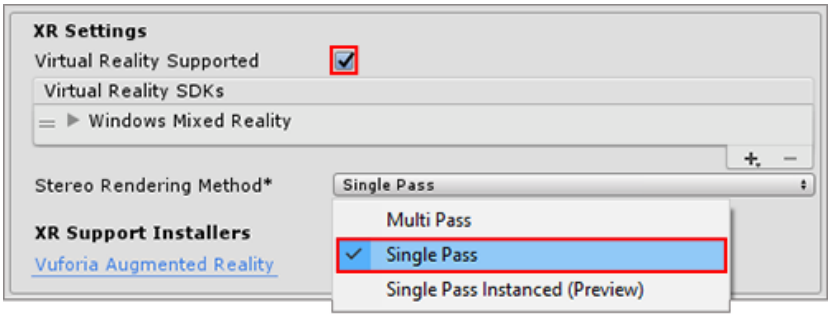
Normal VR rendering:

Left-eye image on the left, right-eye image on the right

Single-Pass Stereo VR rendering:

Left-eye and right-eye images packed together

To enable this feature, open **PlayerSettings** (menu: **Edit** > **Project Settings** > **Player**). In **PlayerSettings**, navigate to **XR Settings**, ensure the **Virtual Reality Supported** checkbox is ticked, then select the **Single Pass** option from the **Stereo Rendering Method** dropdown.



Selecting Single Pass rendering from **PlayerSettings > XR Settings**

Unity's built-in rendering features and **Standard Assets** all support this feature. However, custom-built **Shaders** and **Shaders** downloaded from the **Asset Store** might need to be modified (for example, you might need to scale and offset screen space coordinates to access the appropriate half of the packed Render Texture) to add Single Pass Stereo rendering support.

Adding Single Pass Stereo rendering support to Shaders

Existing helper methods in `UnityCG.cginc` support Single Pass Stereo rendering. Whether your application is **XR** or not, you still need to perform transformations on vertices. For example, when creating any type of application, the vertices enter the **vertex shader** stage in model space and exit in clip space. The **vertex shader** must output clip space vertex coordinates. The set of vertices affected by the shader normally begins in model space before the **vertex shader** transforms them into clip space. However, in order for these vertices to arrive in clip space, the **vertex shader** first transforms them into world space and then to **viewport** space.

In the case of XR, there are multiple view matrices: one for both the left and right eye. You can use the built-in method `UnityWorldToClipPos` to ensure that Unity takes into consideration whether the calculation requires handling multiple view matrices. If you use the`UnityWorldToClipPos` method, the shader automatically performs the transformation calculation correctly, regardless of the platform your application is running on.

`UnityCG.cginc` also contains the following helper methods to assist you with authoring stereoscopic Shaders:

Property	Parameters	Description
UnityStereoScreenSpaceUVAdjust(uv, sb)	`uv`: UV texture coordinates. Either a float2 for a standard UV or a float4 for a packed pair of two UVs. `sb`: A float4 containing a 2D scale and 2D bias which the shader applies to the UV, with scale in xy and bias in zw.	If `UNITY_SINGLE_PASS_STEREO` is defined, this returns the result of applying the scale and bias in sb to the texture coordinates in uv. Otherwise, this returns the texture coordinates unmodified. Use this to apply a per-eye scale and bias only when in Single Pass Stereo rendering mode.
UnityStereoTransformScreenSpaceTex(uv)	`uv`: UV texture coordinates. Either a float2 for a standard UV or a float4 for a packed pair of two UVs.	If `UNITY_SINGLE_PASS_STEREO` is defined, this returns the result of applying the current eye's scale and bias to the texture coordinates in uv. Otherwise, this returns the texture coordinates unaltered.
UnityStereoClamp(uv, sb)	`uv`: UV texture coordinates. Either a float2 for a standard UV or a float4 for a packed pair of two UVs. `sb`: A float4 containing a 2D scale and 2D bias which the shader applies to the UV, with scale in xy and bias in zw.	If `UNITY_SINGLE_PASS_STEREO` is defined, this returns the result of applying the clamp to the x value, using the width, and bias, provided by sb, to the texture's coordinates in uv. Otherwise returns the texture coordinates unmodified. Use this to apply a per-eye clamping in Single Pass Stereo rendering mode to avoid color bleeding between eyes.

Shaders expose the constant in-built variable 'unity_StereoEyeIndex', so that Unity can perform eye-dependent calculations. The value of `unity_StereoEyeIndex` is 0 for left-eye rendering, and 1 for right-eye rendering.

Here is an example from `UnityCG.cginc`, demonstrating how you can use`unity_StereoEyeIndex` to modify screen space coordinates:

```
float2 TransformStereoScreenSpaceTex(float2 uv, float w)
{
    float4 scaleOffset = unity_StereoScaleOffset[unity_StereoEyeIndex];
    return uv.xy * scaleOffset.xy + scaleOffset.zw * w;
}
```

In most cases, you don't need to modify your Shaders. However, there are situations in which you might need to sample a monoscopic Texture as a source for Single Pass Stereo rendering (for example, if you are creating a full-screen film grain or noise effect where the source image should be the same for both eyes, rather than packed into a stereoscopic image). In such situations, use `ComputeNonStereoScreenPos()` instead of `ComputeScreenPos()` to calculate locations from the full source Texture.

Post-processing effects

Post-processing effects require some extra work to support Single Pass Stereo rendering. Each post-processing effect runs once on the packed Render Texture (which contains both the left and right eye images), but applies all draw commands that run during the post-processing twice: once to the left-eye half of the destination Render Texture, and once to the right-eye half.

Post-processing effects do not automatically detect Single Pass Stereo rendering, so you need to adjust any reads of packed Stereo Render Textures so that they only read from the correct side for the eye being rendered. There are two ways to do this depending on how your post-processing effect is being rendered:

Using Graphics.Blit()
Mesh-based drawing

Without the above-mentioned adjustments, each draw command reads the whole of the source Render Texture (containing both the left and right eye views), and outputs the entire image pair to both the left and right eye sides of the output Render Texture, resulting in incorrect duplication of the source image in each eye.

This happens when using `Graphics.Blit` or a fullscreen polygon with a Texture map to draw each post-processing effect . Both methods reference the entire output of the previous post-processing effect in the chain. When referring to an area in a packed stereo Render Texture, they reference the whole packed Render Texture instead of just the relevant half of it.

Graphics.Blit()

Post-processing effects rendered with `Blit()` do not automatically reference the correct part of packed stereo Render Textures. By default, they refer to the entire texture. This incorrectly stretches the post-processing effect across both eyes.

For Single Pass Stereo rendering using `Blit()`, texture samplers in Shaders have an additional auto-calculated variable which refers to the correct half of a packed stereo Render Texture, depending on the eye being drawn. The variable contains scale and offset values that allow you to transform your target coordinates to the correct location.

To access this variable, declare a `half4` in your Shader with the same name as your sampler, and add the `suffix _ST` (see below for a code example of this). To adjust UV coordinates, pass in your `_ST` variable to `scaleAndOffset` and use `UnityStereoScreenSpaceUVAdjust(uv, scaleAndOffset)`. This method compiles to nothing in non-Single Pass Stereo builds, meaning that shaders modified to support this mode are still compatible with non-Single Pass Stereo builds.

The following examples demonstrate what you need to change in your fragment shader code to support Single Pass Stereo rendering.

Without stereo rendering:

```
uniform sampler2D _MainTex;

fixed4 frag (v2f_img i) : SV_Target
{
    fixed4 myTex = tex2D(_MainTex, i.uv);
    ...
}
```

With stereo rendering:

```
uniform sampler2D _MainTex;
half4 _MainTex_ST;

fixed4 frag (v2f_img i) : SV_Target
{
    fixed4 myTex = tex2D(_MainTex, UnityStereoScreenSpaceUVAdjust(i.uv, _MainTex_ST));
```

```
    ...
}
```

Mesh-based drawing

Rendering Post-processing effects using meshes (for example, by drawing a quadrilateral in immediate mode using the low level graphics API) also need to adjust the UV coordinates on the target Texture when rendering each eye. To adjust your coordinates in these circumstances, use `UnityStereoTransformScreenSpaceTex(uv)`. This method correctly adjusts for packed stereo Render Textures in Single Pass Stereo rendering mode, and automatically compiles for non-packed Render Textures if you have disabled Single Pass Stereo rendering mode. However, if you intend to use a Shader both for packed and unpacked Render Textures in the same mode, you need to have two separate Shaders.

Screen space effects

Screen space effects are visual effects that are drawn over a pre-rendered image. Examples of screen space effects include ambient occlusion, depth of field, and bloom.

For example, imagine a screen space effect that requires an image to be drawn over the screen (perhaps you are drawing some kind of dirt spattered on the screen). Instead of applying the effect over the entire output display, which would stretch the dirt image across both eyes, you need to apply it twice: once for each eye. In cases like this, you need to convert from using texture coordinates that reference the whole packed Render Texture, to coordinates that reference each eye.

The following code examples show a Surface Shader that repeats an input Texture (called **_Detail**) 8 x 6 times across the output image. In the second example, the shader transforms the destination coordinates in Single Pass Stereo mode to refer to the part of the output Texture that represents the eye currently being rendered.

Example 1: Detail Texture with no Single-Pass Stereo support

```
void surf(Input IN, inout SurfaceOutput o)
{
    o.Albedo = tex2D(_MainTex, IN.uv_MainTex).rgb;
    float2 screenUV = IN.screenPos.xy / IN.screenPos.w;
    screenUV *= float2(8,6);
    o.Albedo *= tex2D(_Detail, screenUV).rgb * 2;
}
```

Example 2: Detail Texture with Single-Pass Stereo support

```
void surf(Input IN, inout SurfaceOutput o)
{
    o.Albedo = tex2D(_MainTex, IN.uv_MainTex).rgb;
    float2 screenUV = IN.screenPos.xy / IN.screenPos.w;
    #if UNITY_SINGLE_PASS_STEREO
    // If Single-Pass Stereo mode is active, transform the
    // coordinates to get the correct output UV for the current eye.
    float4 scaleOffset = unity_StereoScaleOffset[unity_StereoEyeIndex];
    screenUV = (screenUV - scaleOffset.zw) / scaleOffset.xy;
    #endif
    screenUV *= float2(8,6);
    o.Albedo *= tex2D(_Detail, screenUV).rgb * 2;
}
```

2018–08–16 Page published with limited editorial review

New in Unity 2017.3

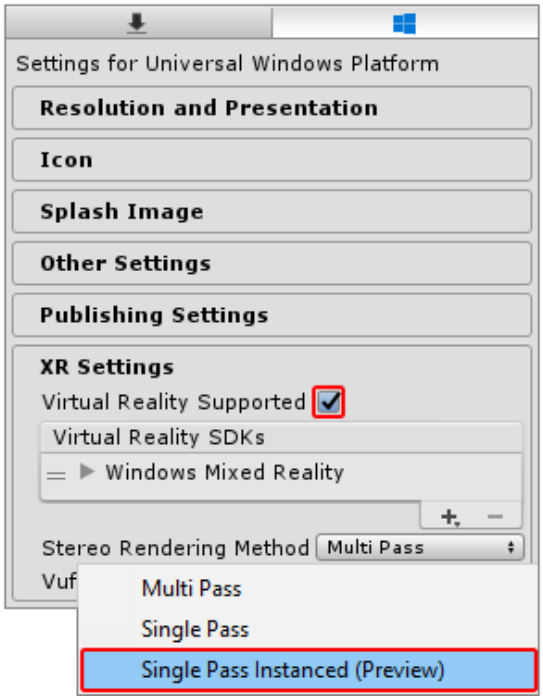
Single Pass Instanced rendering

Leave feedback

With **Single Pass Instanced rendering** (also known as Stereo Instancing), the GPU performs a single render pass, replacing each draw call with an instanced draw call. This heavily decreases CPU use, and slightly decreases GPU use, due to the cache coherency between the two draw calls. This significantly reduces power consumption of your application.

Supported platforms

PlayStation VR
Oculus Rift (DirectX 11)
HoloLens
Magic Leap
Android devices that support the Multiview extension
For DirextX on desktop, the GPU must support Direct3D 11 and the `VPAndRTArrayIndexFromAnyShaderFeedingRasterizer` extension.
For OpenGL on desktop, the GPU must support one of the following extensions:
`GL_NV_viewport_array2`
`GL_AMD_vertex_shader_layer`
`GL_ARB_shader_viewport_layer_array`
To enable this feature, open **Player Settings** (go to **Edit** > **Project Settings** > **Player**). In **Player Settings**, navigate to **XR Settings** at the bottom, check the **Virtual Reality Supported** box, then select **Single Pass Instanced (Preview)** from the **Stereo Rendering Method** drop-down menu.



In **XR Settings**, set **Stereo Rendering Method** to **Single Pass Instanced (Preview)**

The default **Stereo Rendering Method** is **Multi Pass**. This setting is slower, but usually works better with custom **shaders**. If you have custom **shaders**, you might need to change them to make them compatible with **Single Pass Instanced** rendering.

Custom shaders

Before you follow the instructions below, update your custom shaders to use instancing (see GPU Instancing).

Next, you need to make two additional changes in the last shader stage used before the **fragment shader** (Vertex/Hull/Domain/Geometry) for any of your custom shaders.

For each custom shader you want to support Single Pass Instancing, carry out the following steps:

Step 1: Add `UNITY_VERTEX_INPUT_INSTANCE_ID` to the `appdata struct`.

Example:

```
struct appdata
{
    float4 vertex : POSITION;
    float2 uv : TEXCOORD0;

    UNITY_VERTEX_INPUT_INSTANCE_ID //Insert
};
```

Step 2: Add `UNITY_VERTEX_OUTPUT_STEREO` to the `v2f output struct`.

Example:

```
struct v2f
{
    float2 uv : TEXCOORD0;
    float4 vertex : SV_POSITION;

    UNITY_VERTEX_OUTPUT_STEREO //Insert
};
```

Step 3: Add the `UNITY_SETUP_INSTANCE_ID()` macro at the beginning of your main `vert` method, followed by a call to `UNITY_INITIALIZE_OUTPUT(v2f, o)` and `UNITY_INITIALIZE_VERTEX_OUTPUT_STEREO()`.

`UNITY_SETUP_INSTANCE_ID()` calculates and sets the built-in `unity_StereoEyeIndex` and `unity_InstanceID` Unity shader variables to the correct values based on which eye the GPU is currently rendering.

`UNITY_INITIALIZE_VERTEX_OUTPUT_STEREO` tells the GPU which eye in the texture array it should render to, based on the value of `unity_StereoEyeIndex`. This macro also transfers the value of `unity_StereoEyeIndex` from the **vertex shader** so that it will be accessible in the fragment shader only if `UNITY_SETUP_STEREO_EYE_INDEX_POST_VERTEX` is called in the fragment shader `frag` method.

`UNITY_INITALIZE_OUTPUT(v2f,o)` initializes all `v2f` values to 0.

Example:

```
v2f vert (appdata v)
{
    v2f o;

    UNITY_SETUP_INSTANCE_ID(v); //Insert
    UNITY_INITIALIZE_OUTPUT(v2f, o); //Insert
    UNITY_INITIALIZE_VERTEX_OUTPUT_STEREO(o); //Insert

    o.vertex = UnityObjectToClipPos(v.vertex);

    o.uv = v.uv;

    return o;
}
```

Post-Processing shaders

If you want your Post-Processing shaders to support Single Pass Instancing, follow the steps in Custom shaders as well as the steps below . You can download all Unity base shader scripts from the Unity website.

Do the following for each Post-Processing shader that you want to support Single Pass Instancing:

Step 1: Add the UNITY_DECLARE_SCREENSPACE_TEXTURE(tex) macro outside the frag method (see the placement example below) in your Shader script, so that when you use a particular stereo rendering method the GPU uses the appropriate texture sampler. For example, if you use Multi-Pass rendering, the GPU uses a texture 2D sampler. For single pass instancing or multi-view rendering, the texture sampler is a texture array.

Step 2: Add `UNITY_SETUP_STEREO_EYE_INDEX_POST_VERTEX(i)` at the beginning of the fragment shader frag method (See the placement example below). You only need to add this macro if you want to use the `unity_StereoEyeIndex` built-in shader variable to find out which eye the GPU is rendering to. This is useful when testing post processing effects.

Step 3: Use the `UNITY_SAMPLE_SCREENSPACE_TEXTURE()` macro when sampling 2D textures (See the placement example below). Standard shaders use a 2D texture-based back buffer to sample textures. Single Pass Stereo Instancing does not use this type of back buffer, so if you do not specify a different method for 2D texture sampling, your shader does not render correctly. To prevent rendering issues, the `UNITY_SAMPLE_SCREENSPACE_TEXTURE()` macro detects which stereo rendering path you are using and then automatically samples the texture in the correct manner. See Unity documentation on HLSLSupport.cginc to learn more about similar macros used for depth textures and screen-space shadow maps.

Example:

```
UNITY_DECLARE_SCREENSPACE_TEXTURE(_MainTex); //Insert

fixed4 frag (v2f i) : SV_Target
{
    UNITY_SETUP_STEREO_EYE_INDEX_POST_VERTEX(i); //Insert
```

```
    fixed4 col = UNITY_SAMPLE_SCREENSPACE_TEXTURE(_MainTex, i.uv); //Insert

    // just invert the colors

    col = 1 - col;

    return col;
}
```

Full sample shader code

Below is a simple example of the template image effect shader with all of the previously mentioned changes applied to allow Single Pass Instancing support. The additions to the shader code are marked with a comment (`//Insert`).

```
struct appdata
{
    float4 vertex : POSITION;
    float2 uv : TEXCOORD0;

    UNITY_VERTEX_INPUT_INSTANCE_ID //Insert
};

//v2f output struct

struct v2f
{

    float2 uv : TEXCOORD0;
    float4 vertex : SV_POSITION;

    UNITY_VERTEX_OUTPUT_STEREO //Insert
};

v2f vert (appdata v)
{
    v2f o;

    UNITY_SETUP_INSTANCE_ID(v); //Insert
    UNITY_INITIALIZE_OUTPUT(v2f, o); //Insert
    UNITY_INITIALIZE_VERTEX_OUTPUT_STEREO(o); //Insert

    o.vertex = UnityObjectToClipPos(v.vertex);
    o.uv = v.uv;
    return o;
}

UNITY_DECLARE_SCREENSPACE_TEXTURE(_MainTex); //Insert
```

```
fixed4 frag (v2f i) : SV_Target
{
    UNITY_SETUP_STEREO_EYE_INDEX_POST_VERTEX(i); //Insert

    fixed4 col = UNITY_SAMPLE_SCREENSPACE_TEXTURE(_MainTex, i.uv); //Insert

    // invert the colors

    col = 1 - col;

    return col;
}
```

Procedural geometry

When using the Graphics.DrawProceduralIndirect() and CommandBuffer.DrawProceduralIndirect() methods to draw fully procedural geometry on the GPU, it is important to note that both methods receive their arguments from a compute buffer. This means that it is difficult to increase the instance count at run time. To increase the instance count, you need to manually double the instance count contained in your compute buffers.

See Vertex and fragment shader examples for more information on how to write shader code.

Debugging your shader

The following shader code renders a GameObject as green for a user's left eye and red for their right eye. This shader is useful for debugging your stereo rendering, because it allows you to verify that all stereo graphics work and are functioning correctly.

```
Shader "__XR__/StereoEyeIndexColor"
{
   Properties
   {
       _LeftEyeColor("Left Eye Color", COLOR) = (0,1,0,1)
       _RightEyeColor("Right Eye Color", COLOR) = (1,0,0,1)
   }

   SubShader
   {
      Tags { "RenderType" = "Opaque" }

      Pass
      {
         CGPROGRAM

         #pragma vertex vert
         #pragma fragment frag
```

```
            float4 _LeftEyeColor;
            float4 _RightEyeColor;

            #include "UnityCG.cginc"

            struct appdata
            {
               float4 vertex : POSITION;

               UNITY_VERTEX_INPUT_INSTANCE_ID
            };

            struct v2f
            {
               float4 vertex : SV_POSITION;

               UNITY_VERTEX_INPUT_INSTANCE_ID
               UNITY_VERTEX_OUTPUT_STEREO
            };

            v2f vert (appdata v)
            {
               v2f o;

               UNITY_SETUP_INSTANCE_ID(v);
               UNITY_INITIALIZE_OUTPUT(v2f, o);
               UNITY_INITIALIZE_VERTEX_OUTPUT_STEREO(o);

               o.vertex = UnityObjectToClipPos(v.vertex);

               return o;
            }

            fixed4 frag (v2f i) : SV_Target
            {
               UNITY_SETUP_STEREO_EYE_INDEX_POST_VERTEX(i);

               return lerp(_LeftEyeColor, _RightEyeColor, unity_StereoEyeIndex);
            }
            ENDCG
         }
      }
   }
```

2018–09–07 Page published with limited editorial review

Single Pass Instancing new in 2017.3 2017.3

Single-Pass Stereo Rendering for HoloLens

Leave feedback

There are two stereo **rendering** methods for **Windows Holographic** devices (HoloLens); multi-pass and single-pass instanced.

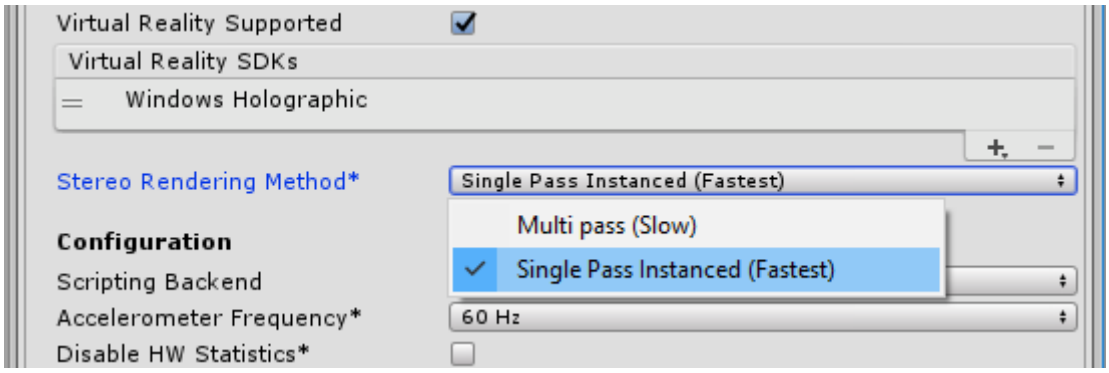
Multi-pass

Multi-pass rendering runs 2 complete render passes (one for each eye). This generates almost double the CPU workload compared to the single-pass instanced rendering method. However this method is the most backwards compatible and doesn't require any **shader** changes.

Single-pass Instanced

Instanced rendering performs a single render pass where each draw call is replaced with an instanced draw call. This heavily decreases CPU utilization. Additionally this slightly decreases GPU utilization due to the cache coherency between the two draw calls. In turn your app's power consumption will be much lower.

To enable this feature, open **PlayerSettings** (menu: **Edit** > **Project Settings** > **Player**). In **PlayerSettings**, navigate to **Other Settings**, check the **Virtual Reality Supported** checkbox, then select **Single Pass Instanced (Fastest)** from the **Stereo Rendering Method** dropdown.



Unity defaults to the slower **Multi pass (Slow)** setting as you may have custom shaders that do not have the required code in your **scripts** to support this feature.

Shader script requirements

Any non built-in shaders will need to be updated to work with instancing. Please read this documentation to see how this is done: GPU Instancing. Furthermore, you'll need to make two additional changes in the last shader stage used before the **fragment shader** (Vertex/Hull/Domain/Geometry). First, you will have to add `UNITY_VERTEX_OUTPUT_STEREO` to the output struct. Second, you will need to add `UNITY_INITIALIZE_VERTEX_OUTPUT_STEREO()` in the main function for that stage after `UNITY_SETUP_INSTANCE_ID()` has been called.

Post-Processing Shaders

You will need to add the `UNITY_DECLARE_SCREENSPACE_TEXTURE(tex)` macro around the input texture declarations, so that 2D texture arrays will be properly declared. Next, you must add a call to `UNITY_SETUP_INSTANCE_ID()` at the beginning of the fragment shader. Finally, you will need to use the

`UNITY_SAMPLE_SCREENSPACE_TEXTURE()` macro when sampling those textures. See HLSLSupport.cginc for more information on other similar macros depth textures and screen space shadow maps.

Here's a simple example that applies all of the previously mentioned changes to the template image effect:

```
struct appdata
{
    float4 vertex : POSITION;
    float2 uv : TEXCOORD0;
    UNITY_INSTANCE_ID
};
struct v2f
{
    float2 uv : TEXCOORD0;
    float4 vertex : SV_POSITION;
    UNITY_INSTANCE_ID
    UNITY_VERTEX_OUTPUT_STEREO
};
v2f vert (appdata v)
{
    v2f o;
    UNITY_SETUP_INSTANCE_ID(v);
    UNITY_TRANSFER_INSTANCE_ID(v, o);
    UNITY_INITIALIZE_VERTEX_OUTPUT_STEREO(o);
    o.vertex = UnityObjectToClipPos(v.vertex);
    o.uv = v.uv;
    return o;
}

UNITY_DECLARE_SCREENSPACE_TEXTURE(_MainTex);

fixed4 frag (v2f i) : SV_Target
{
    UNITY_SETUP_INSTANCE_ID(i);
    fixed4 col = UNITY_SAMPLE_SCREENSPACE_TEXTURE(_MainTex, i.uv);
    // just invert the colors
    col = 1 - col;
    return col;
}
```

DrawProceduralIndirect

`Graphics.DrawProceduralIndirect()` and `CommandBuffer.DrawProceduralIndirect()` get all of their arguments from a compute buffer, so we can't easily increase the instance count. Therefore you will have to manually double the instance count contained in your compute buffers.

See the Vertex and fragment shader examples page for more information on shader code.

2017–09–01 Page amended with limited editorial review

New feature in 5.5

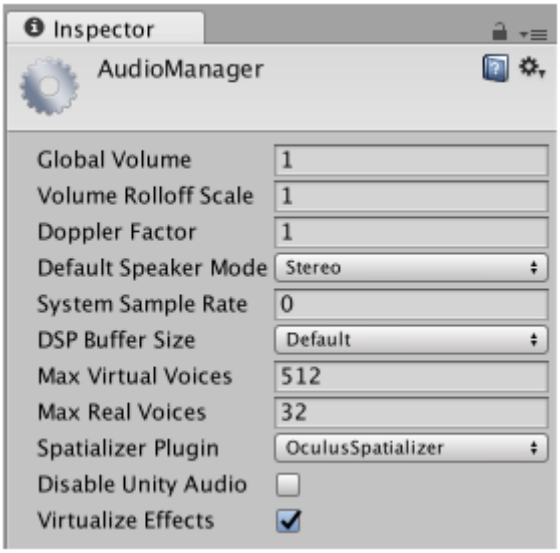
Fixed example code for single pass stereo rendering for HoloLens

VR Audio Spatializers

Leave feedback

Unity natively supports Audio Spatializers for **virtual reality** (VR) projects. Audio Spatializers change the way audio is transmitted from an **audio source** into the surrounding space: the **plugin** takes the source and regulates the gains of the left and right ear contributions based on the distance and angle between the AudioListener and the AudioSource.

Enable these plugins through the Audio Settings Window (menu: **Edit > Project Settings > Audio**) using the **Spatializer Plugin** dropdown. The native plugins can be used with or without VR mode enabled.



The spatializer plugins only work on the platforms that a VR device is supported on. If a device is not supported for a build target, Unity displays a warning that the plugin will not be included in the built application.

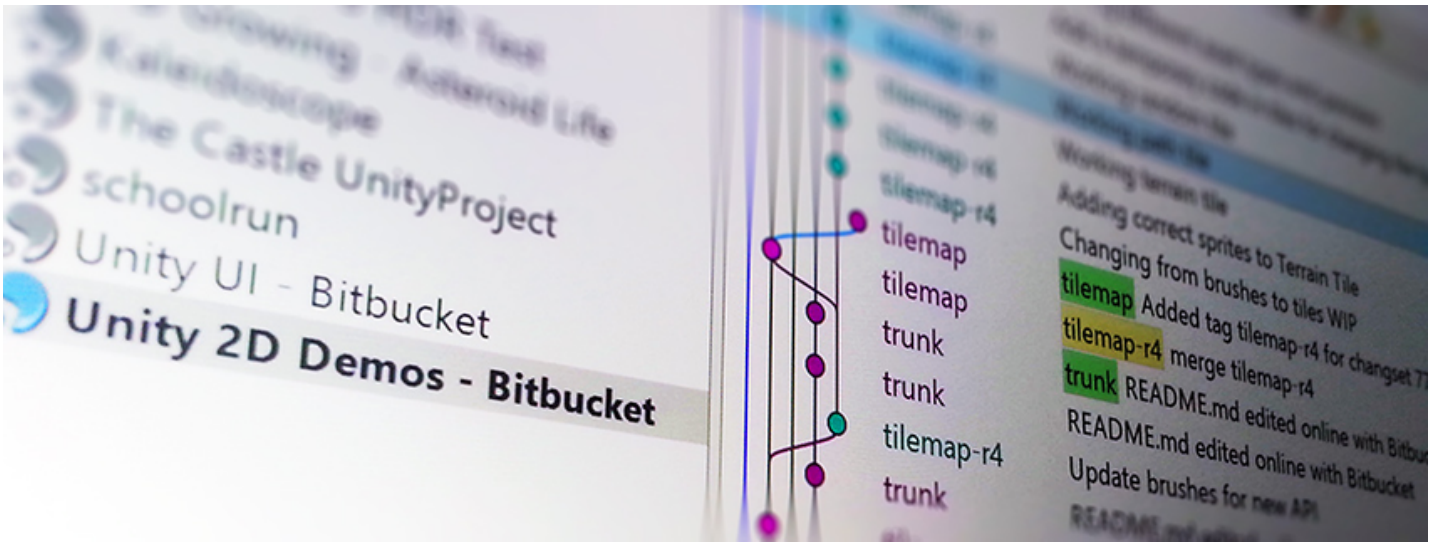
Every plugin works in the Editor for testing purposes.

Natively included Spatializer plugins:

Oculus Spatializer (supports Android, OSX, and PC)
Microsoft HRTF Spatializer (supports UWP and PC running Windows 10)

Open-source repositories

Leave feedback



Our decision to start open-sourcing **components** of Unity is intended to help us engage with you, our customers and users. We want to provide you with all of the securities and flexibilities that having source access provides.

We always welcome collaborative participation and community development. It is our hope and expectation that with this initiative, our community will be able to extend Unity in ways that were previously not possible.

Unity repositories

We recommend that you use Bitbucket when contributing to Unity, as this is what we tend to use internally, and it keeps things simple. Unity has a number of open-source repositories to help developers use the Unity software:

http://bitbucket.org/Unity-Technologies/

This is our newest repository, used primarily by the UnityUI team and the R&D Content team. Some core R&D projects end up on here (largely due to developer preference), such as the 5.3 Memory **Profiler**.

https://github.com/Unity-Technologies

This is our oldest repository, where some of our core R&D projects keep their open-source work. Key projects include Unity's fork of Mono (the C# runtime), and Katana (our build/CI system). Much of the tech used for **WebGL**, such as Emscripten, is also hosted here.

https://github.com/unity3d-jp

This repository is used by Unity Japan's R&D and Evangelism teams. It contains demo projects, prototypes and side projects from that office's employees. Interesting projects include the upcoming **Asset** Graph system for building AssetBundle, a demonstration project illustrating how to effectively use threads in Unity, and an example project demonstrating how to use Unity Ads.

If you'd like to learn more about contributing to the development of Unity, please see the section on Open Source Contributions.

2018–03–19 Page amended with limited editorial review

MonoDevelop replaced by Visual Studio from 2018.1

How to contribute to Unity

Leave feedback

A guide for open-source contributers

This guide provides help and support for Unity users who may not have participated in an open-source project before.

Unity uses distributed version control to version open-source **components**. Essentially, this means that you make changes and contribute them back through a process of “forking” our repository, cloning your “fork”, pushing your changes to your “fork”, and then opening a pull request for us to review.

This guide includes an example walkthrough to help make this process clearer. Please also see the Further Reading section and the FAQ section to learn more about open-source contributions to Unity.

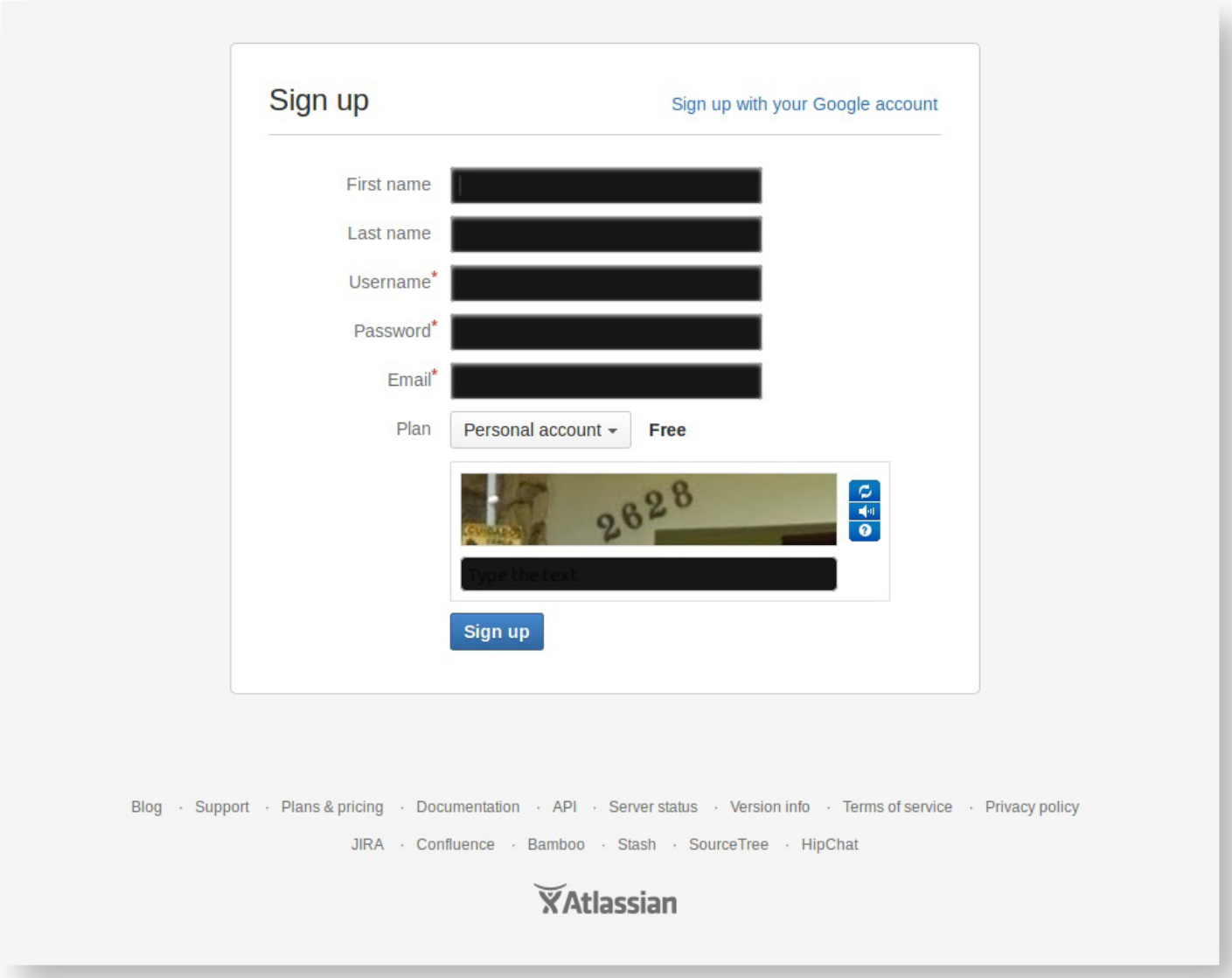
Step 1: Get a Bitbucket account
Step 2: Fork the repository you want to contribute to
Step 3: Clone your fork
Step 4: Apply modifications to your fork
Step 5: Open a pull request on Bitbucket
Step 6: Wait for feedback
Further Reading
FAQ

Step 1: Get a Bitbucket account

Leave feedback

Our open-source **components** are hosted on Bitbucket under the Unity-Technologies team. You don't need to be logged in to access the repositories and clone the source code, but you will need an account in order to be able to contribute your changes.

To get started, visit Bitbucket's account sign-up page.



If you don't want to make a new account, Bitbucket supports OpenID, which will let you log in with your Google, Facebook, Twitter, or GitHub account.

Log in

Switch to OpenID log in

Username or email [redacted]

Password [redacted]

Log in Forgot your password?

Need an account? Sign up free.

or

Google Facebook Twitter GitHub

Blog · Support · Plans & pricing · Documentation · API · Server status · Version info · Terms of service · Privacy policy

JIRA · Confluence · Bamboo · Stash · SourceTree · HipChat

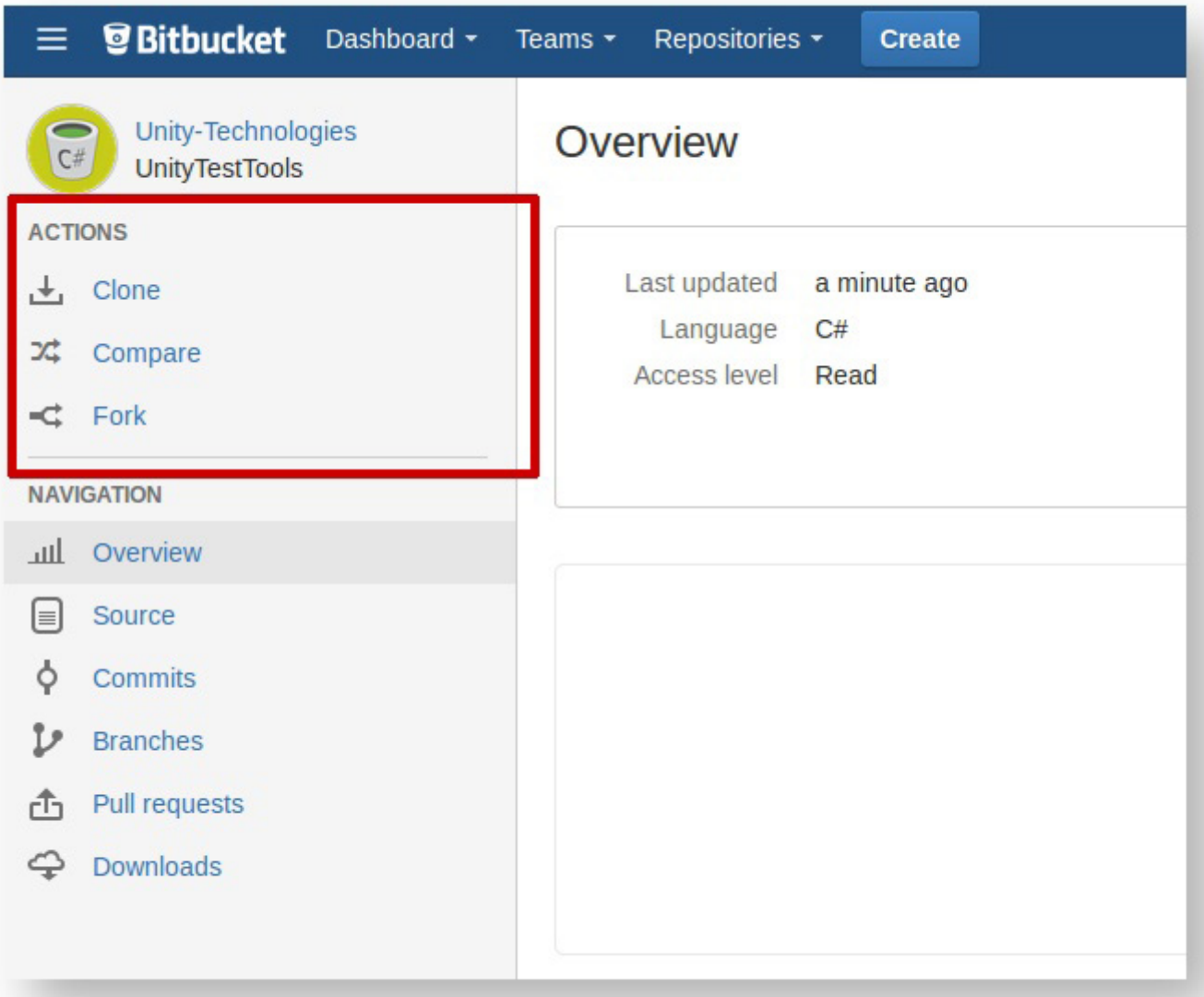
Atlassian

Step 2: Fork the repository you want to contribute to

Leave feedback

This demonstration uses the Unity Test Tools Repo.

There is a sidebar on the left-hand side of the repository page, with an **Actions** section:



Click the **Fork** button and fill out the form you are redirected to, then click **Fork repository**:

Fork Unity-Technologies / UnityTestTools

You can also create a patch queue

Name* UnityTestTools

Description

It's encouraged to write a little about why you are forking.

Access level ☐ This is a private repository

Permissions ☐ Inherit repository user/group permissions

Project management ☐ Issue tracking ☑ Wiki

Fork at tip (14abe49)

Fork repository Cancel

What is a fork?

Forking is a great way to contribute to a project even though you don't have write access. Check out our documentation for more details.

Blog · Support · Plans & pricing · Documentation · API · Server status · Version info · Terms of service · Privacy policy

JIRA · Confluence · Bamboo · Stash · SourceTree · HipChat

Atlassian

When you have finished forking the repo, you'll be taken to the landing page for your new fork:

Bitbucket Dashboard Teams Repositories Create owner/repository

natosha_bard UnityTestTools

ACTIONS

Clone

Create branch

Create pull request

Compare

Fork

NAVIGATION

Overview

Source

Commits

Branches

Pull requests

Wiki

Downloads

Settings

Overview

HTTPS https://natosha_bard@bitbucket.org Share

Last updated just now

Fork of Unity-Technologies/UnityTestTools

Language C#

Access level Admin

1 Branch

1 Tag

0 Forks

1 Watcher

Invite users to this repo

Send invitation

Recent activity

natosha_bard/UnityTestTools

Repository forked from UnityTestTools

NaTosha Bard · just now

There isn't a README yet

Create one and tell people where to start and how to contribute.

Create a README

Blog · Support · Plans & pricing · Documentation · API · Server status · Version info · Terms of service · Privacy policy

JIRA · Confluence · Bamboo · Stash · SourceTree · HipChat

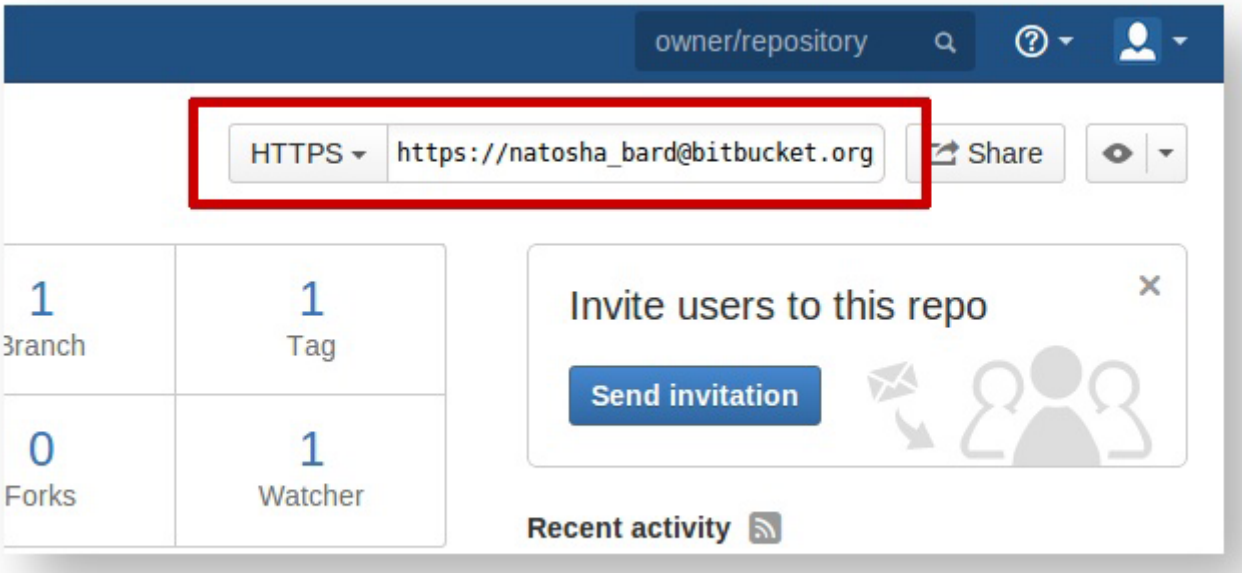
Atlassian

Step 3: Clone your fork

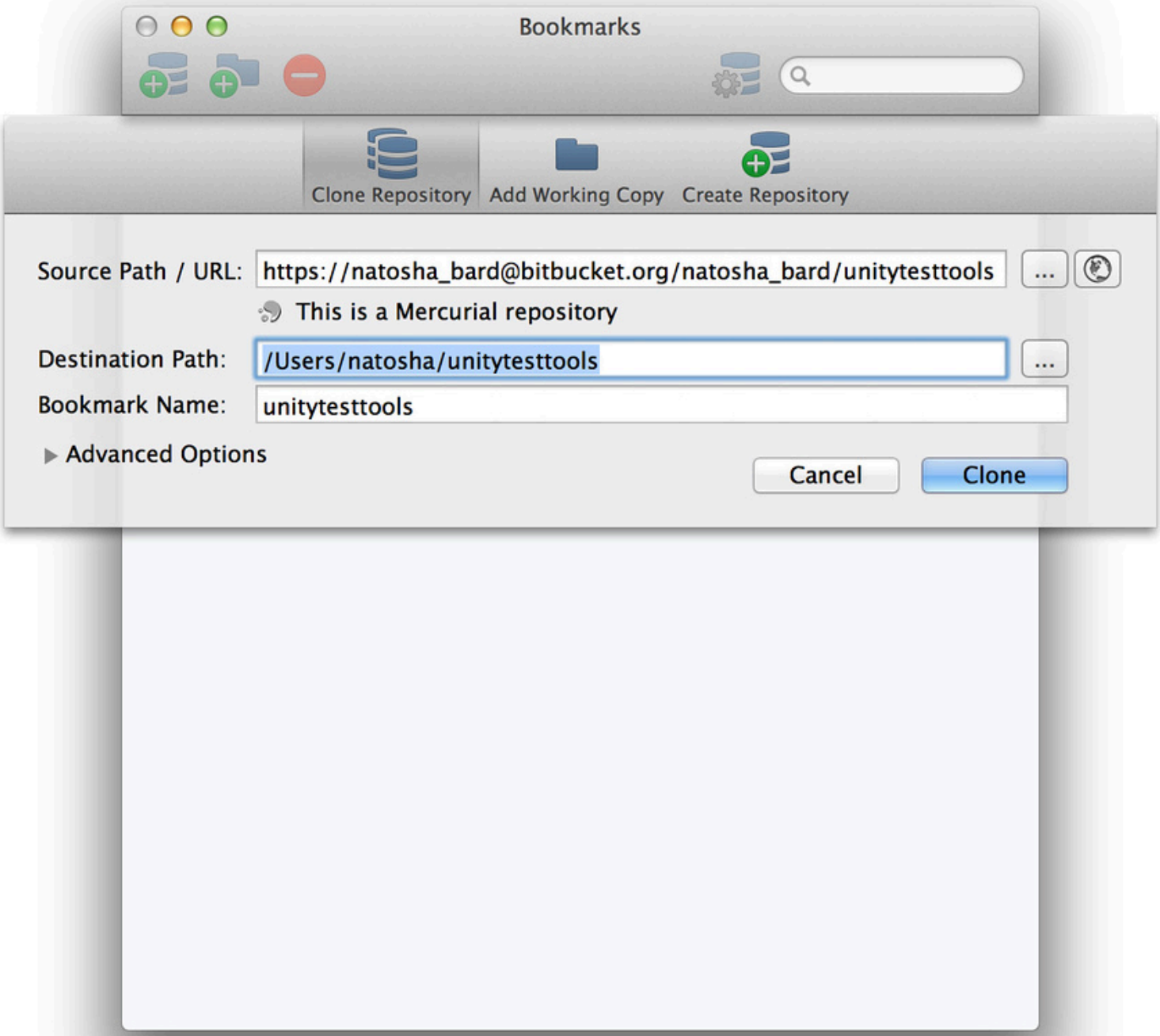
Leave feedback

Next, clone your fork locally so that you can make your modifications. If the repository is a Mercurial repository, you'll need to get a Mercurial client. If it's a Git repository, you'll need to get a Git client. Atlassian (the company that owns Bitbucket) maintains an application called SourceTree, which runs on both Windows and OS X and works with both Mercurial and Git repositories. We use SourceTree heavily within Unity to interact with our own Mercurial repositories. Explore the download page for Mercurial and Git to see the different options available.

Once you have your Mercurial or Git client ready, you can clone it via the URL on your fork's page:



This an example of cloning our fork in SourceTree:

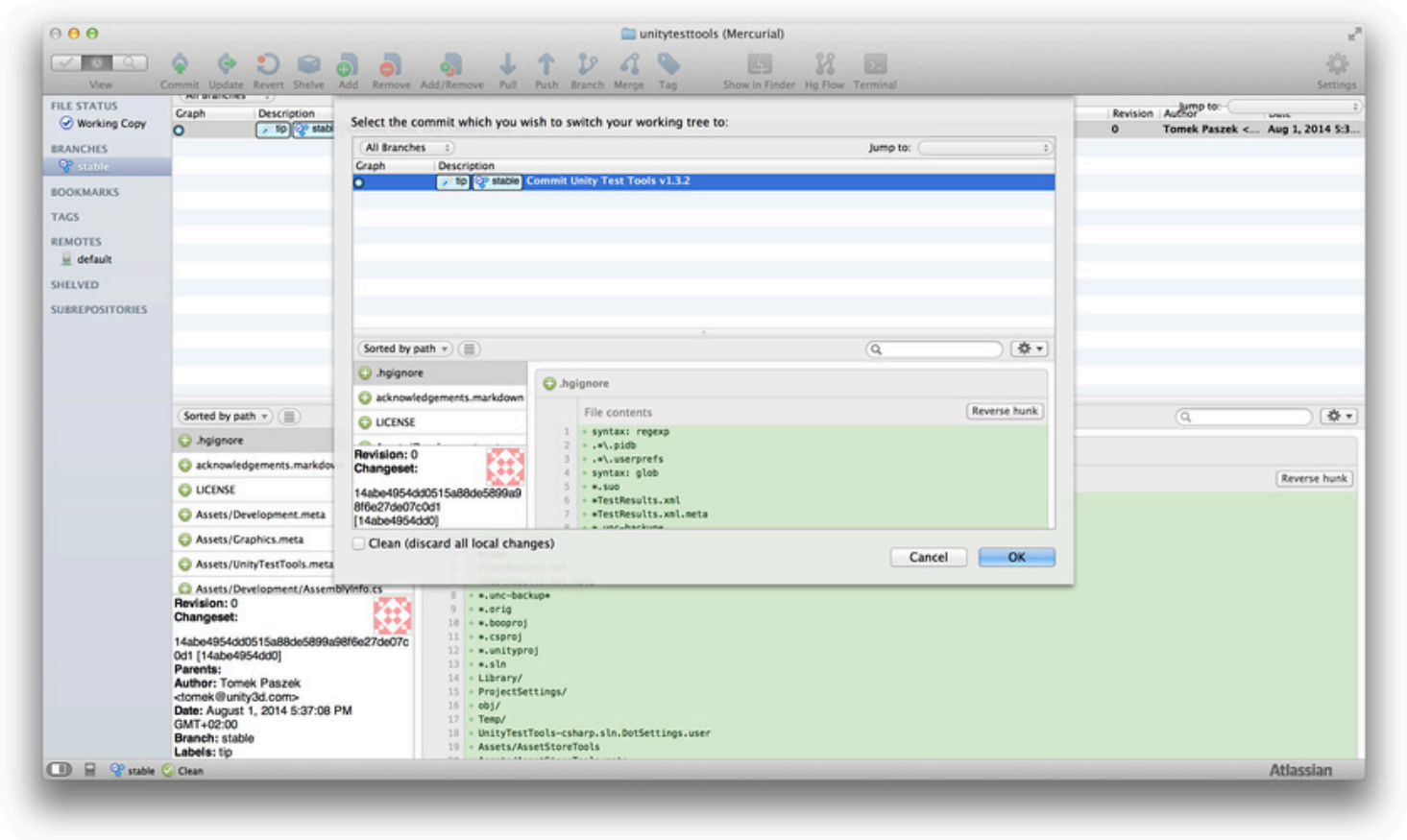

Bookmarks
Clone Repository
Add Working Copy
Create Repository
Source Path / URL:
https://natosha_bard@bitbucket.org/natosha_bard/unitytesttools
This is a Mercurial repository
Destination Path:
/Users/natosha/unitytesttools
Bookmark Name:
unitytesttools
Advanced Options
Cancel
Clone

Step 4: Apply modifications to your fork

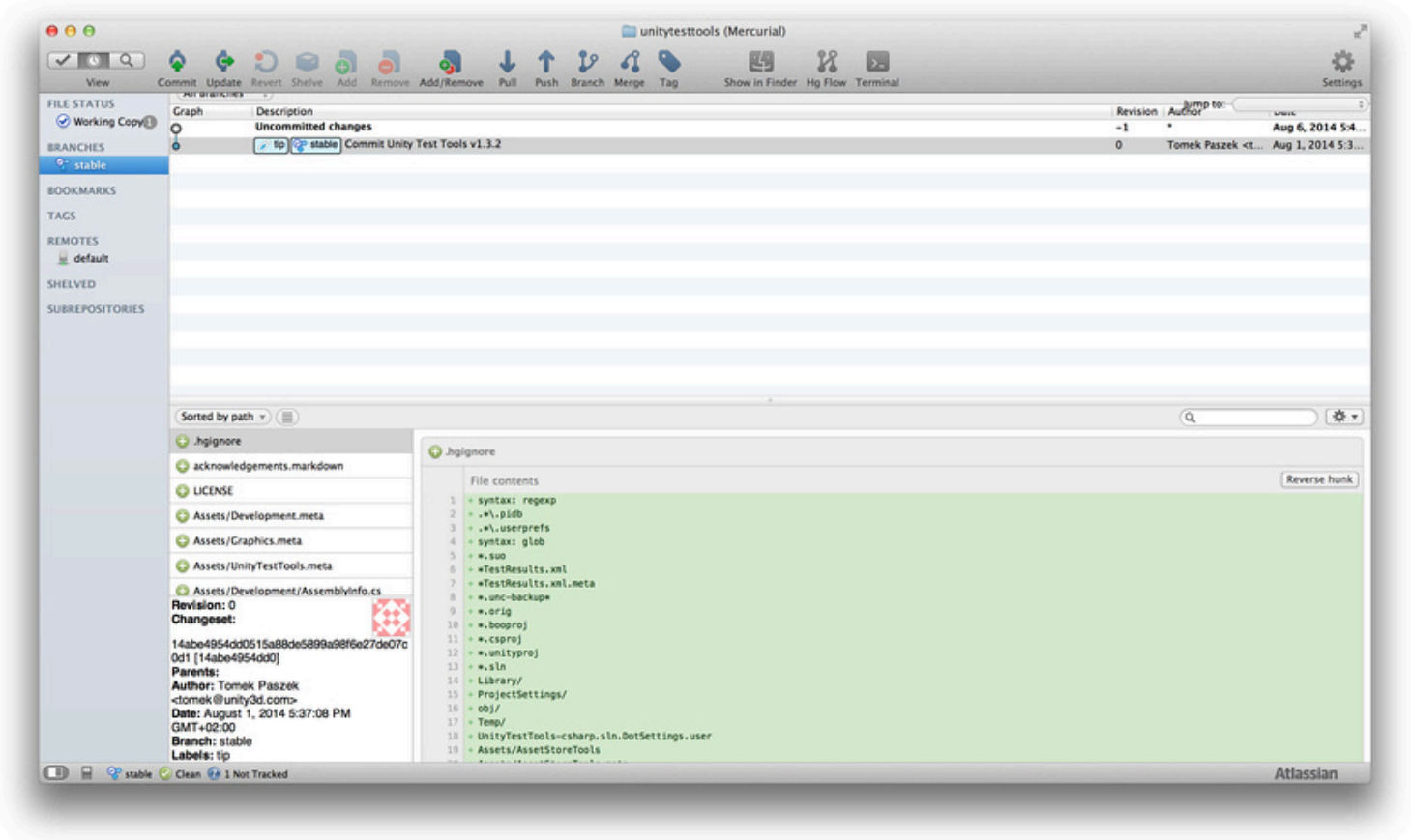
Leave feedback

You should now have a local clone. To make change, the first thing you need to do is access the correct version of the source code by navigating to the correct branch in your Mercurial/Git client. Then you can make your modifications.

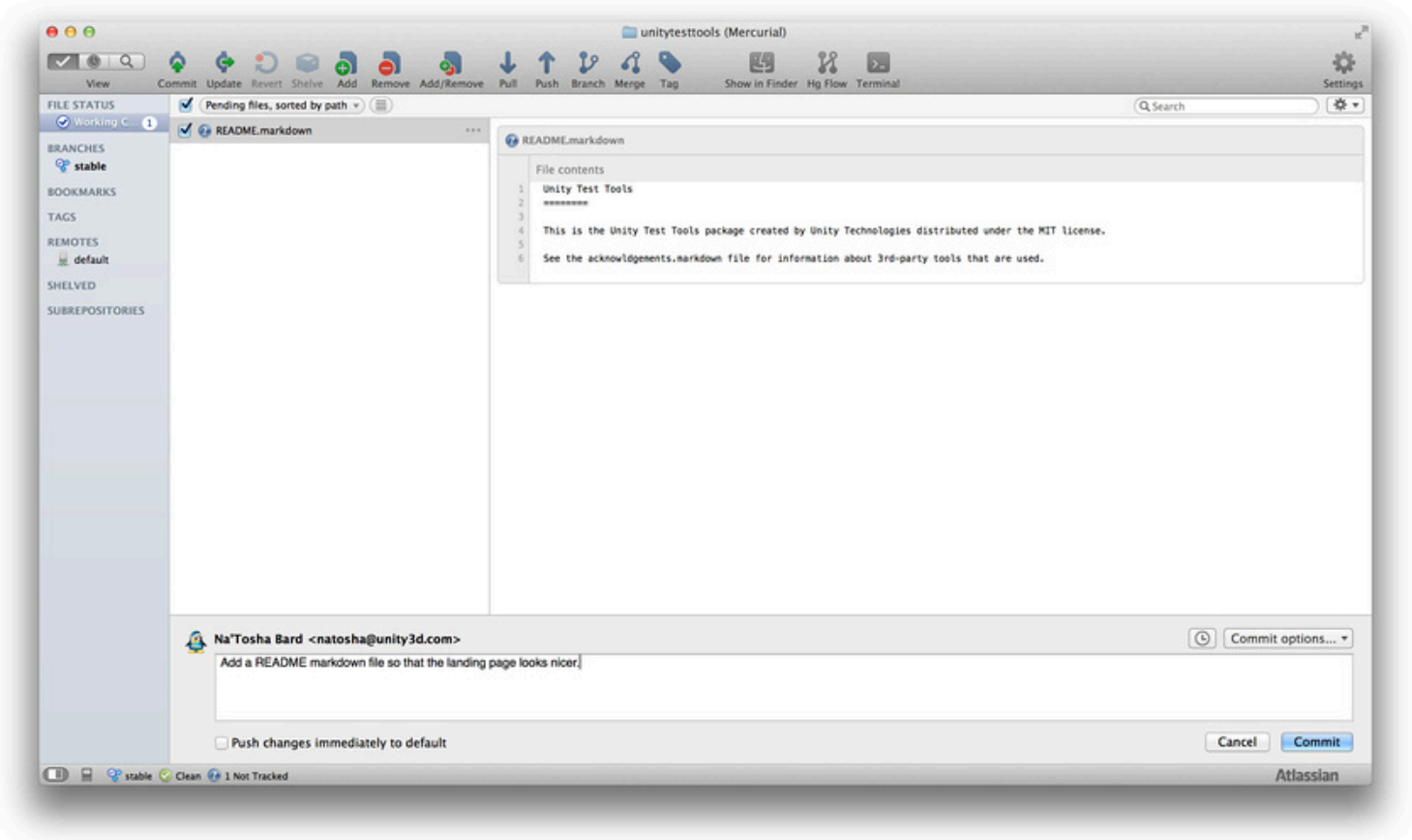
In our example, the Unity Test Tools repo is new and has only one commit on the stable branch. Here is an example of updating explicitly to that version:



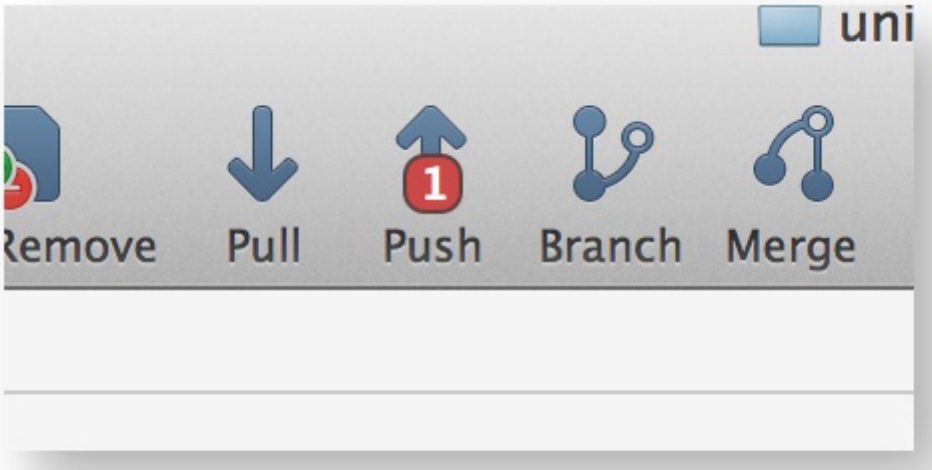
Once you have the correct version of the source code, you can make your modifications as you see fit. When you go back to your Mercurial/Git client, it will detect that you have modified files:



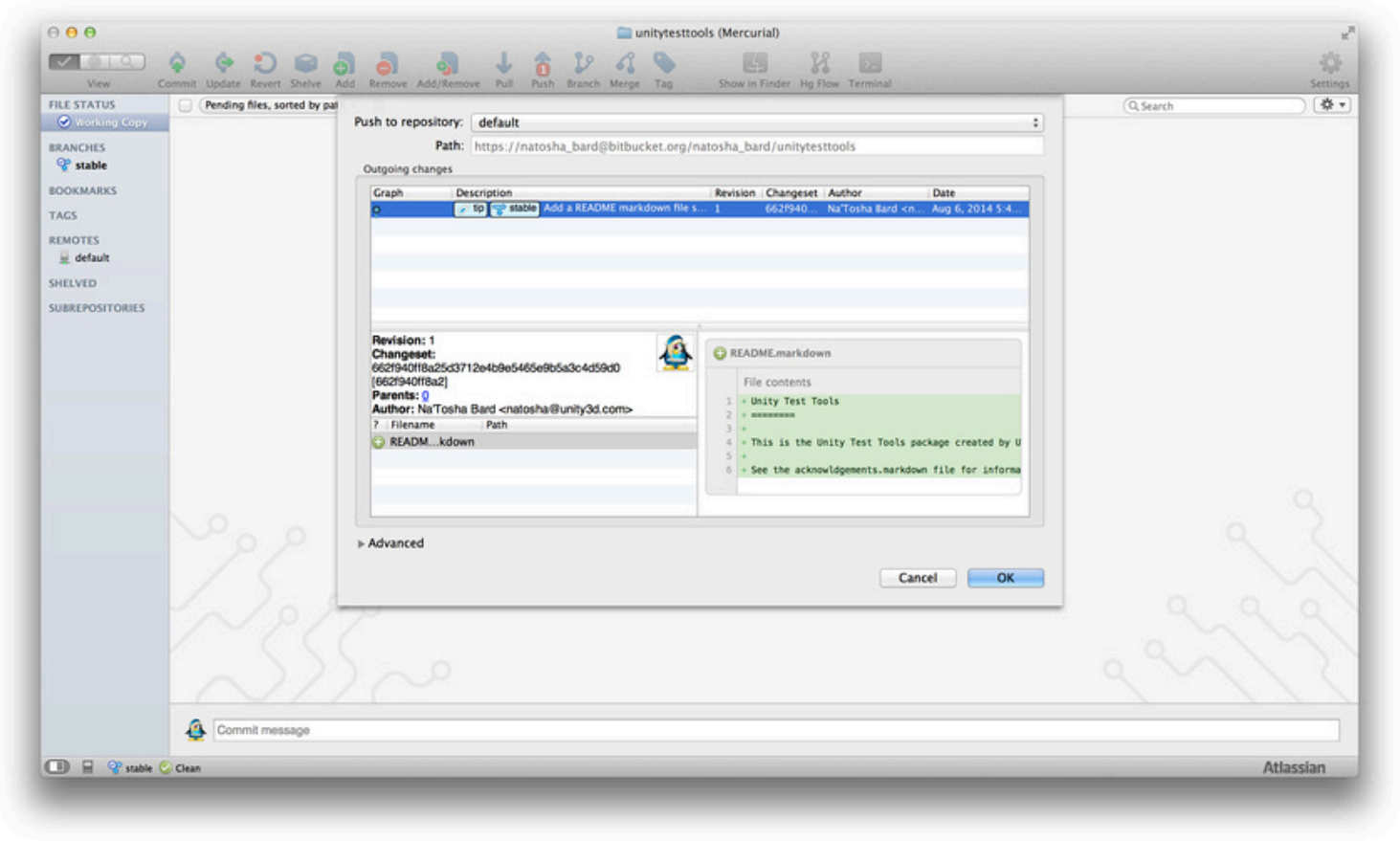
You can see them in the **Working Copy** view. Click **Commit** to commit these changes:



Next, push the changes. Your client will show you how many changesets you have waiting to be pushed:



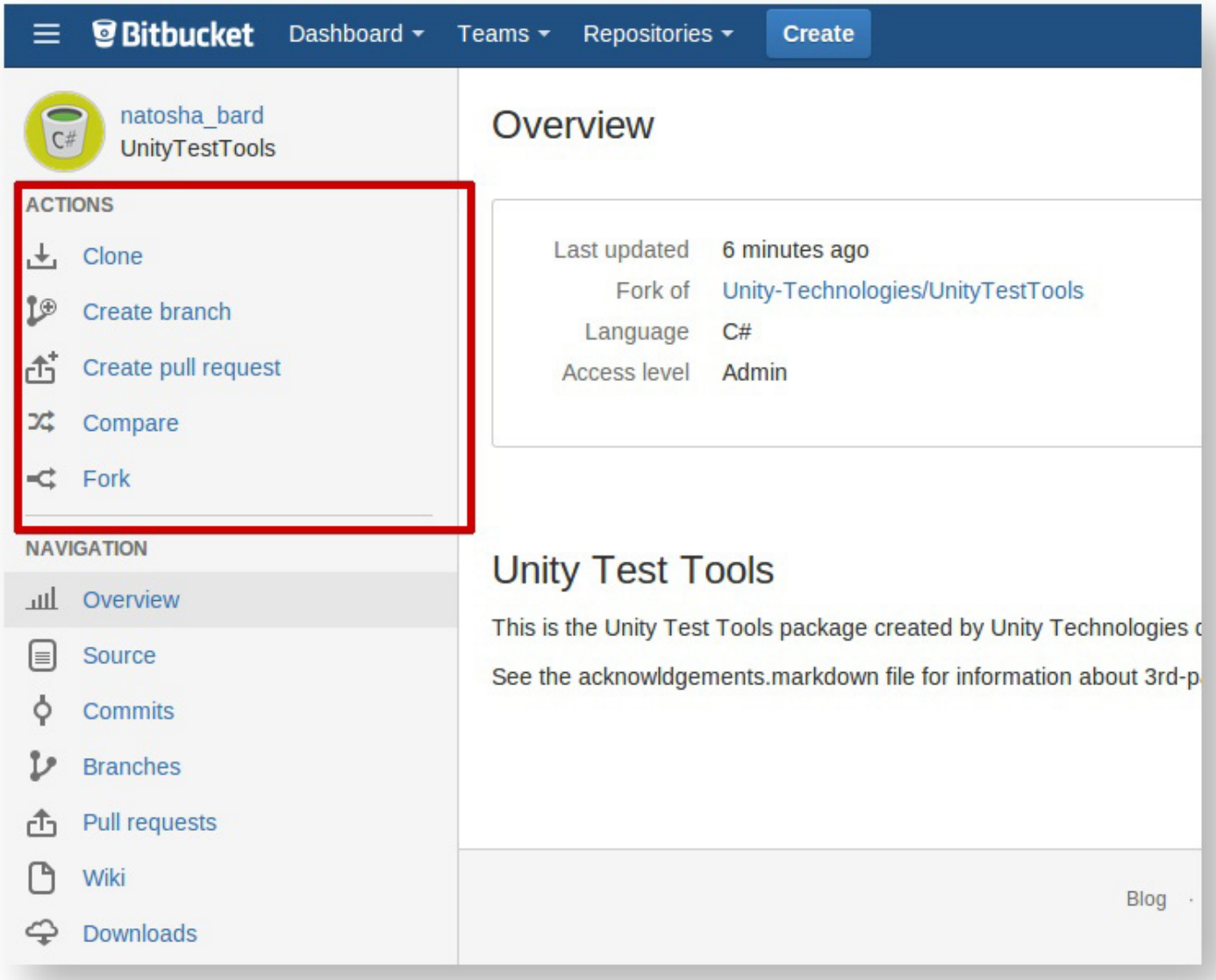
Click **Push** to push your changes:



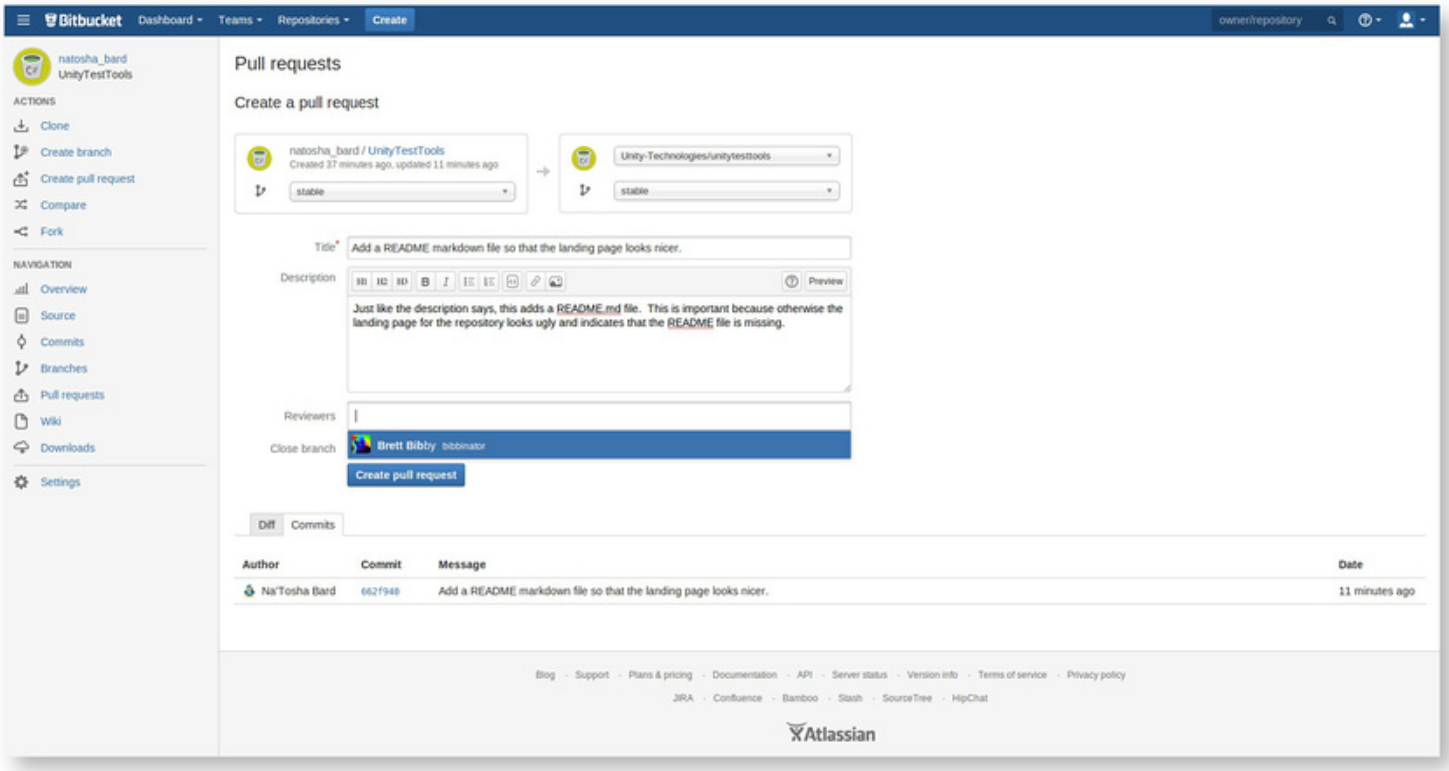
Step 5: Open a pull request on Bitbucket

Leave feedback

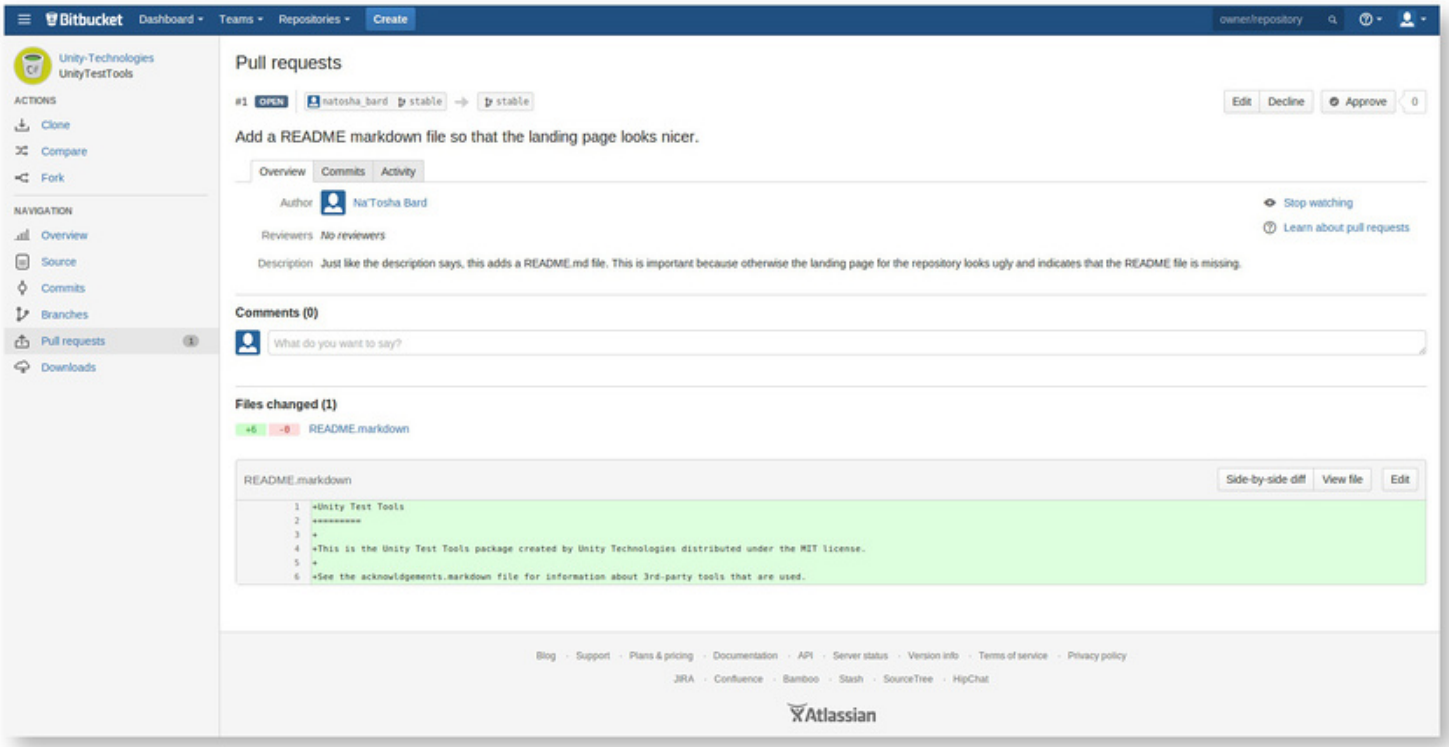
The final step is to open a pull request on Bitbucket. Go back to the landing page for our fork and look at the sidebar again. You will see some new options:



This time, click the **Create Pull Request** link. You'll be taken to a page that automatically summarises the outgoing changesets and shows you a diff of them (a 'diff' shows the differences or changes between two versions). You can also add a description (a good idea!) and any reviewers:



Finally, you'll be redirected to your pull request. You can share this link with anyone you want, including others on your own team so that they can contribute:



Step 6: Wait for feedback

Leave feedback

You now need to wait for feedback from the Unity developers, who are monitoring the repository you are contributing changes to and will have been notified automatically of your pull request. They may have questions, concerns or other feedback, and will communicate these through your pull request, so make sure you're set up to receive notifications from Bitbucket. When the change is accepted, the Unity developer will take care of merging your changes into the official repository maintained by Unity.

Further Reading

Leave feedback

The following external references may be helpful for users who are new to open-source projects and repositories:

Bitbucket 101 - Getting started with Bitbucket
Git Getting Started - Getting started with Git
Getting Started with Git and Mercurial - DVCS overview from Atlassian
Hg Init - A tour of some Mercurial basics
SourceTree FAQ - Frequently asked questions related to SourceTree itself

FAQ

Leave feedback

Here, we've compiled some frequently asked questions:

Q: Why Bitbucket instead of GitHub or self-hosted?

A: At Unity, we are fans of both Bitbucket and GitHub. We also self-host and use a third-party hosting solution called Kallithea internally. Ultimately, we chose Bitbucket for our open-source **components** because:

It allows us to let someone else worry about hosting (which lets us focus on what we're good at)
It supports both Mercurial and Git instead of only Git (we are heavy users of Mercurial internally, but also have some Git-based forks of open-source tools we use, so being able to store everything in one place makes more sense)

Q: What license are Unity's open-source components released under?

A: Unity's open-source components are generally released under an MIT/X11 license. Some projects, like Unity Test Tools, use 3rd-party components that are released under a different license. You can see the license information for each project by looking at the LICENSE file in the top-level of the source directory. Information about third-party tools (if any) that are used in the project are described in an **acknowlegements.markdown** file.

Q: Will Unity accept patches? What about licensing?

A: We will absolutely accept patches. The type of patches we will accept depend on the project because different components are in different stages of development by the Unity devleopers. Bug fixes are great candidates for patches. As for new features or large refactorings, it will depend largely on the system in question.

You should be aware that we will only accept contributions that are licensed under an MIT/X11 license. We will also assume the MIT/X11 license applies to the changes in your pull request unless otherwise stated.

Q: What coding standards does Unity use? How do I make sure my pull request isn't rejected due to bad formatting changes?

A: The best rule of thumb is to make sure to follow the formatting and conventions that already exist in the code you are modifying. Most of the repositories use a coding standard that is similar to Microsoft's C#.

Asset Store Publishing

Leave feedback

This section of the User Manual describes how to publish and manage **Assets** in the Unity Asset Store. Use the navigation column on the left-hand side of the page to view topics in this section.

Adding Keywords to Assets

Leave feedback

What are Keywords ?

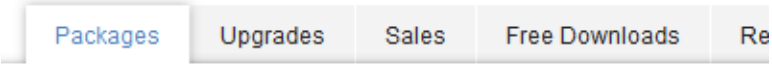
Keywords are just a space delimited list of keywords that publishers use to influence their **asset**'s results in **Asset Store**'s search. There is a limit of 255 characters.

How to add keywords to an asset

Keywords are very simple and easy to add. They are very beneficial for your assets, as they will make them easier for potential customers to locate them within the store when using the search bar.

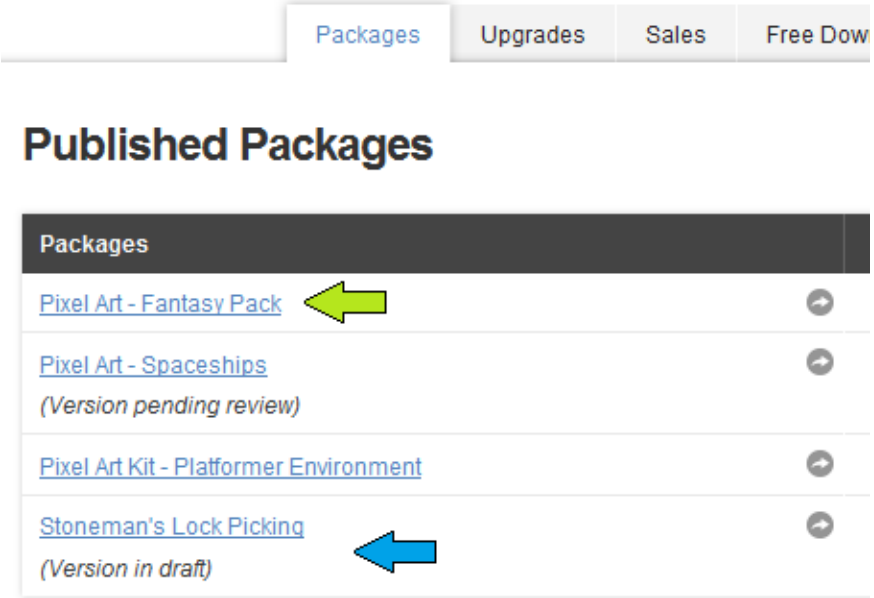
Step 1: Log into your Publisher Admin Area.

Step 2: Navigate to the "Packages" tab.

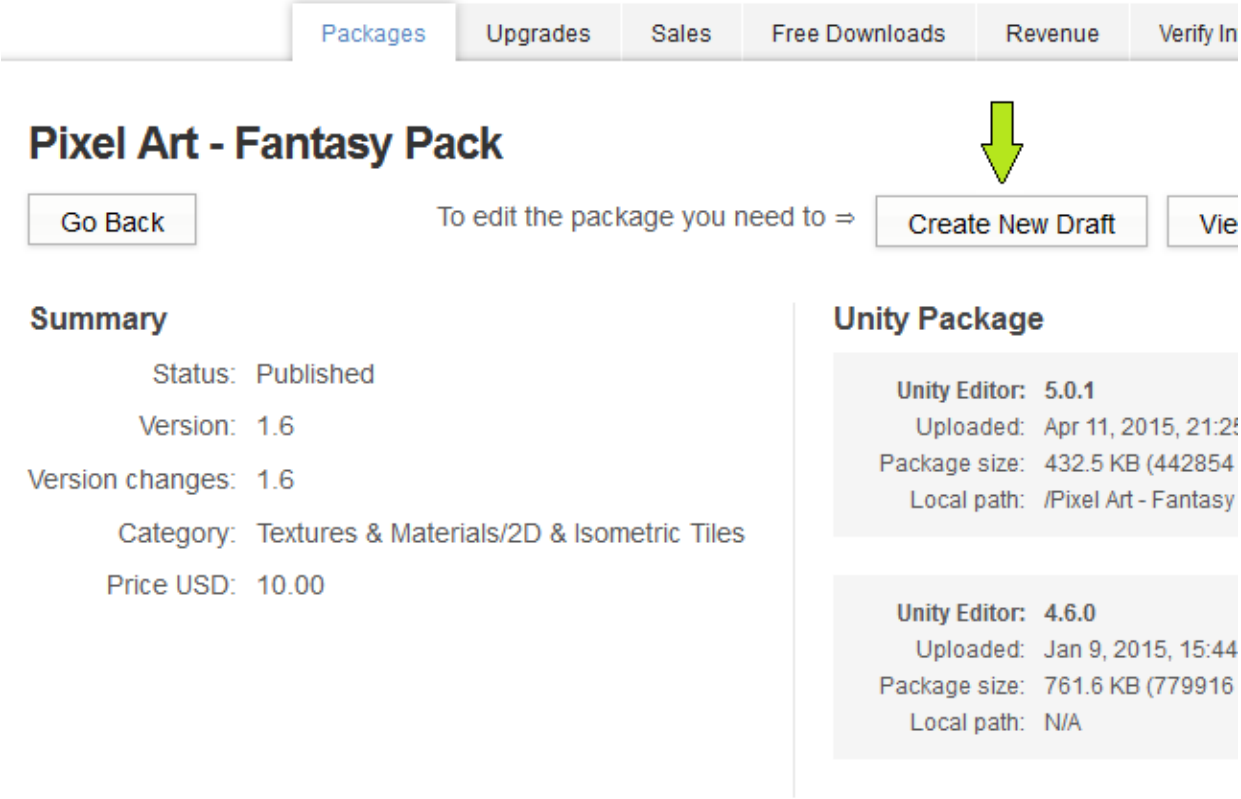


Step 3: Next you will need to create a draft of the asset you wish to add keywords to in order to prepare it for submission and review.

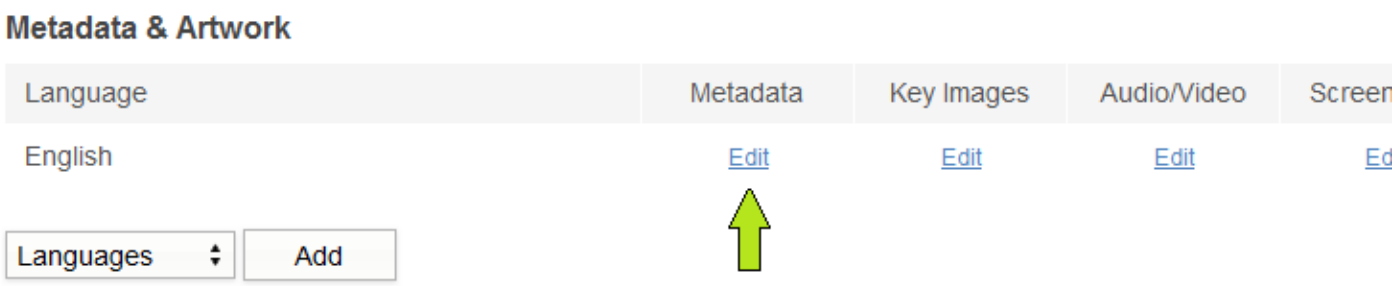
The Blue arrow shows an asset that already has a draft. We will create a draft for the asset indicated by the green arrow.



Step 4: Click the name of the asset to bring up the page below and click the "Create New Draft" button.



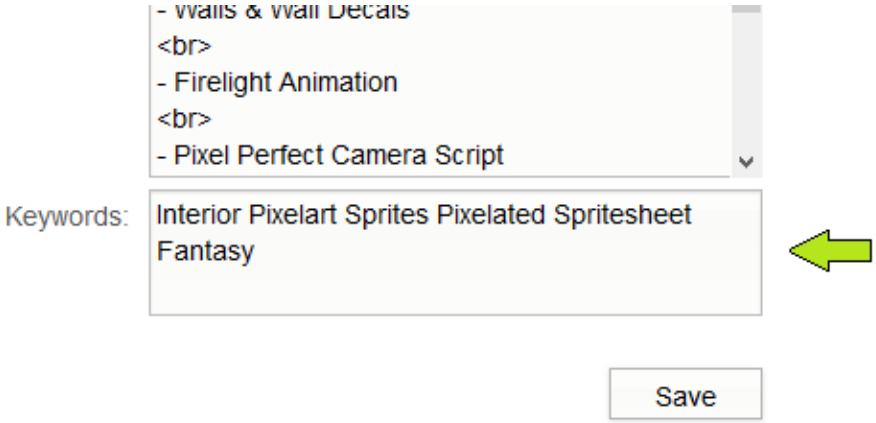
Step 5: Now that we have a draft we can edit the package. Click the Metadata edit button.



Step 6: At the bottom of this page you will be able to enter your Keywords.

Before you, a publisher, add keywords for your asset, think about what kind of search terms you expect developers to use in order to find your asset. Think about how users search. Picking the right keywords in combination with your description, will give you the ability to operate your SEO. If your asset is a 3D Model, maybe using 3D is a waste of a keyword.

Note: Keywords are seperated by whitespace and there is a limit of 255 characters

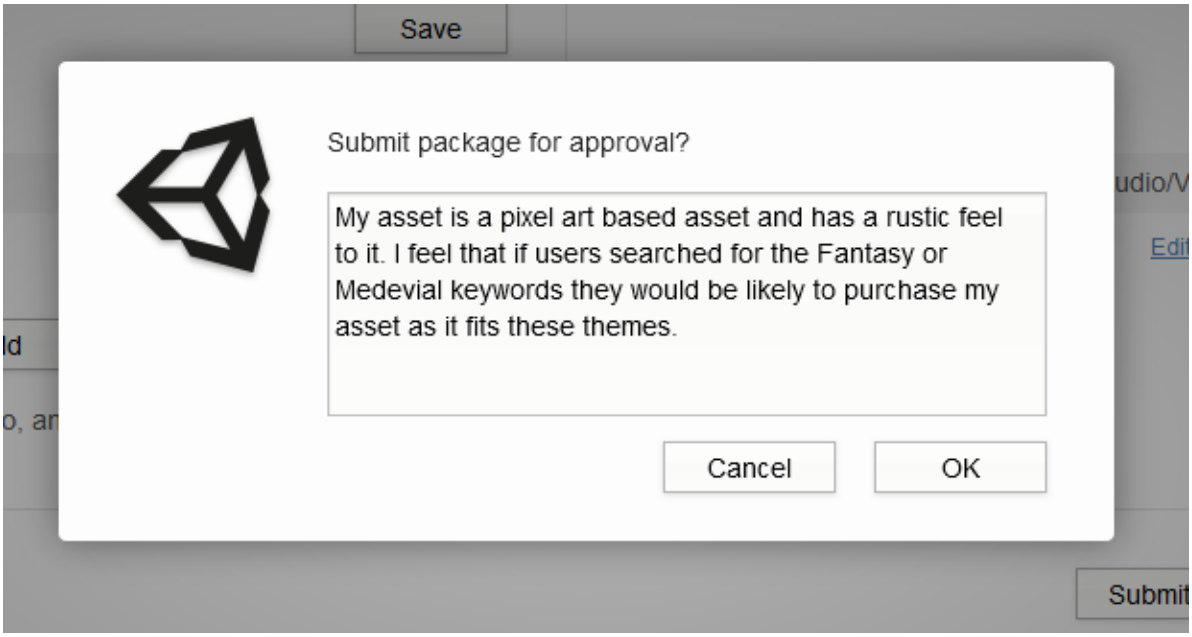


Step 7: Once you have entered all your keywords and are happy with your choices then you can click the “Save” button.

Step 8: When you feel that you are ready to submit. Then return to your draft and Submit the package for approval.

Please add a submission comment (the popup immediately after clicking “Submit package for approval”) for our Asset Store Specialists describing why you chose your keywords. This will be vital information for them to use to make a decision and expedite your update.

Our Asset Store Specialists will vet keywords to ensure that they are relevant to your asset. As there are a lot of publishers, expect around 5 business days for any updates to pass through our process.



Asset Store Publisher Administration

Leave feedback

Setting up a Publisher account

Open the **Asset Store**, sign in, and download the latest version of Asset Store Tools.

Once downloaded and imported to your project, you should see the "**Asset** Store Tools" menu appear in your **Toolbar**. Scroll down and click the **Package Manager** button.

With the Package Manager open, click the link in the top right-hand corner that reads **Publisher account**.

This brings up a window that prompts you to create your Publisher Account. You need to fill out your name, publisher URL, publisher description (including an email support address for your packages) and key images.

Once you're done, hit Save.

Publisher Administration

Once you have your Publisher Account for the Asset Store set up, log into the Publisher Administration portal here: https://publisher.assetstore.unity3d.com/

The Publisher Administration portal contains a number of tabs:

Sales allows you to track all purchases made by customers, organised by package name. Quantity of Sales and Gross Revenue are shown here, as well as any refunds (if applicable).

Free Downloads allows you track all packages you have published on the Asset Store for free, much the same as the Sales tab.

Revenue shows your revenue for any given month since you started selling on the Asset Store. Credits, debits, your balance and recent payouts are shown here.

Pending shows any outstanding packages you have submitted, that are pending approval from our Vetting Team, before being accepted for sale on the Asset Store.

Verify Invoice allows you to verify a purchase using an invoice number - for example, if a customer has purchased on of your assets and needs support with it.

Users allows you to add a number of administrative users (teammates, employees, colleagues) to your master account.

Payout allows you to specify how you would like to receive your earnings. You can amend your payout details for the Asset Store at any point.

Payouts

There are three options when receiving payouts from the Asset Store.

Paypal: No transaction fee. Monthly transfers. No minimum transfer.

Wire: $20 Transaction fee. Quarterly transfers. Minimum transfer of $250.

Check (US only): $15 transaction fee. Quarterly transfers. Minimum transfer of $250.

Check the appropriate box in the **Payout** tab and fill out your details.

Asset Store FAQ

Leave feedback

What date will I receive my monthly or quarterly transfer?

All payouts are scheduled for the 15th of each month

Why haven't I received my payout?

Check your details are correct in the "Payout" tab in your Publisher Administration account. If everything looks correct and you still haven't received your funds, please contact: **assetstore@unity3d.com**

My package has shown as Pending for a while now, what should I do?

Our Vetting Team receive a huge amount of submissions per week, please be patient when waiting for acceptance. If you feel there may be an issue with your submission, please contact **assetstore@unity3d.com** stating your Publisher and Package details.

Can I merge my account with my co-workers' account or another additional account?

Unfortunately not, once an account has been created and purchases have been made, we cannot swap, merge or edit these purchases to appear in another account.

I can't find my answer here. Can I speak to a human about my issue?

Of course you can, drop us an email at **assetstore@unity3d.com**. We aim to respond within 24 hours.

Asset Store Publishing Guide

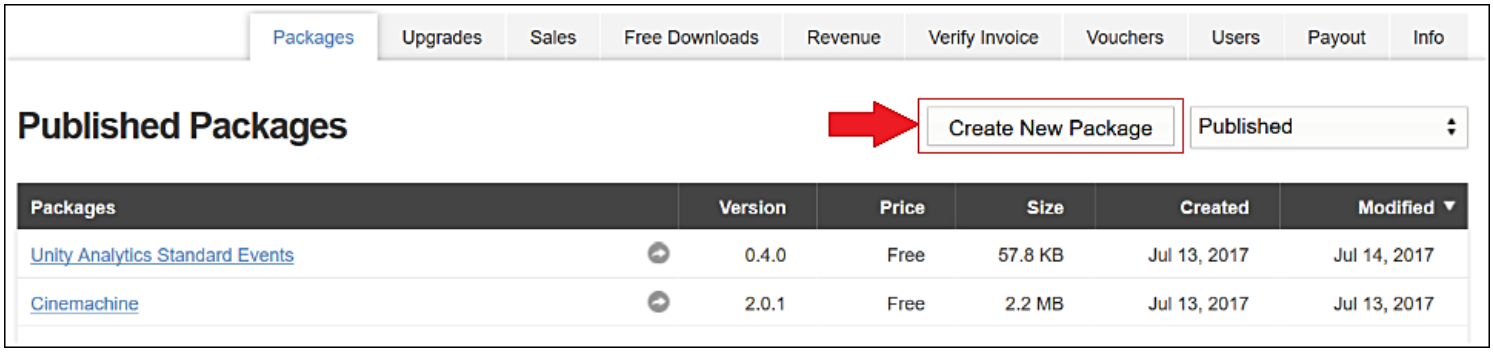
Leave feedback

Creating a Publisher Account

Visit the Publisher Administration page and log in with your Unity Account. If you are not already a Publisher, you will be prompted to create an **Asset Store** Publisher Account. Fill out the form and save the data to create your account. Unless otherwise noted, the information you provide will be displayed on the **Asset Store** on your customer-facing Publisher Page (*Example*).

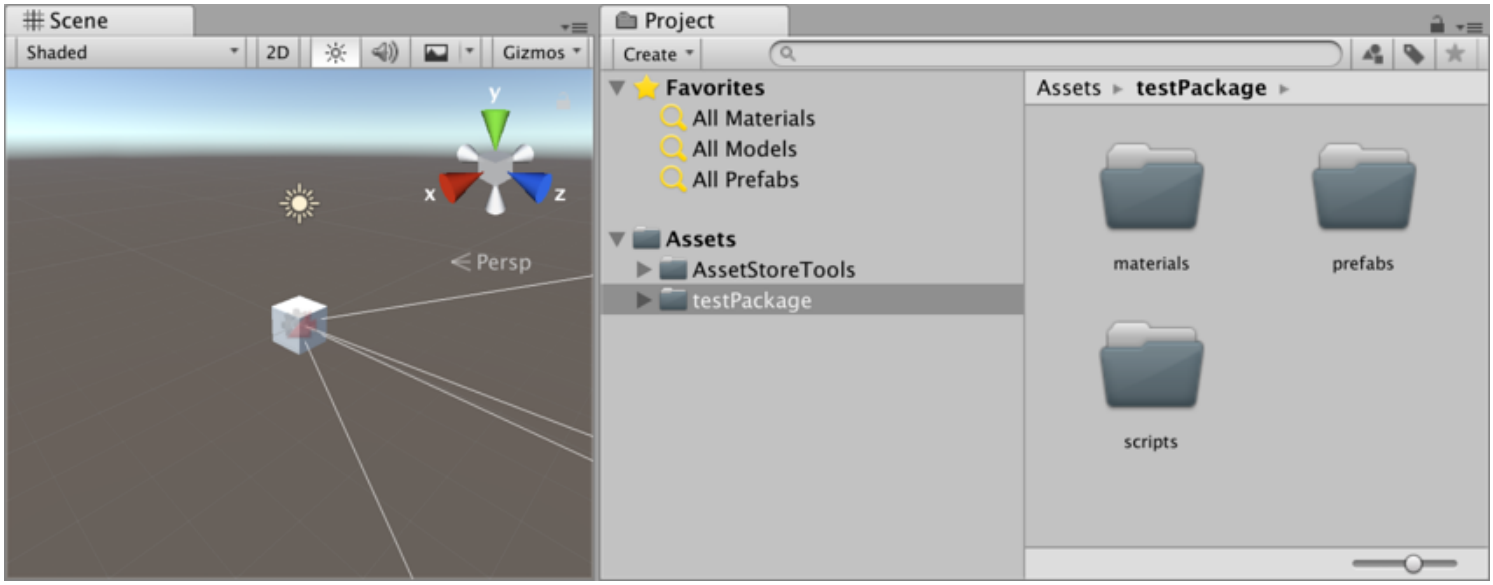
Creating a new package

The first thing you want to do if you are ready to submit an **asset** is to create a draft for your package. Go to the Packages tab of your Publisher Administration page and click the button *"Create New Package"*



Uploading a package

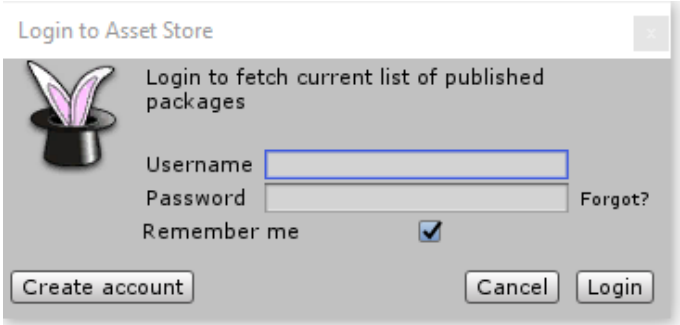
To prevent confusion during upload and import for your users, make sure that your submission content is inside a top-level folder that has the same title as your submission. This folder contains all of the **components** for your Project:



To upload your Asset to the Publisher Administration portal:

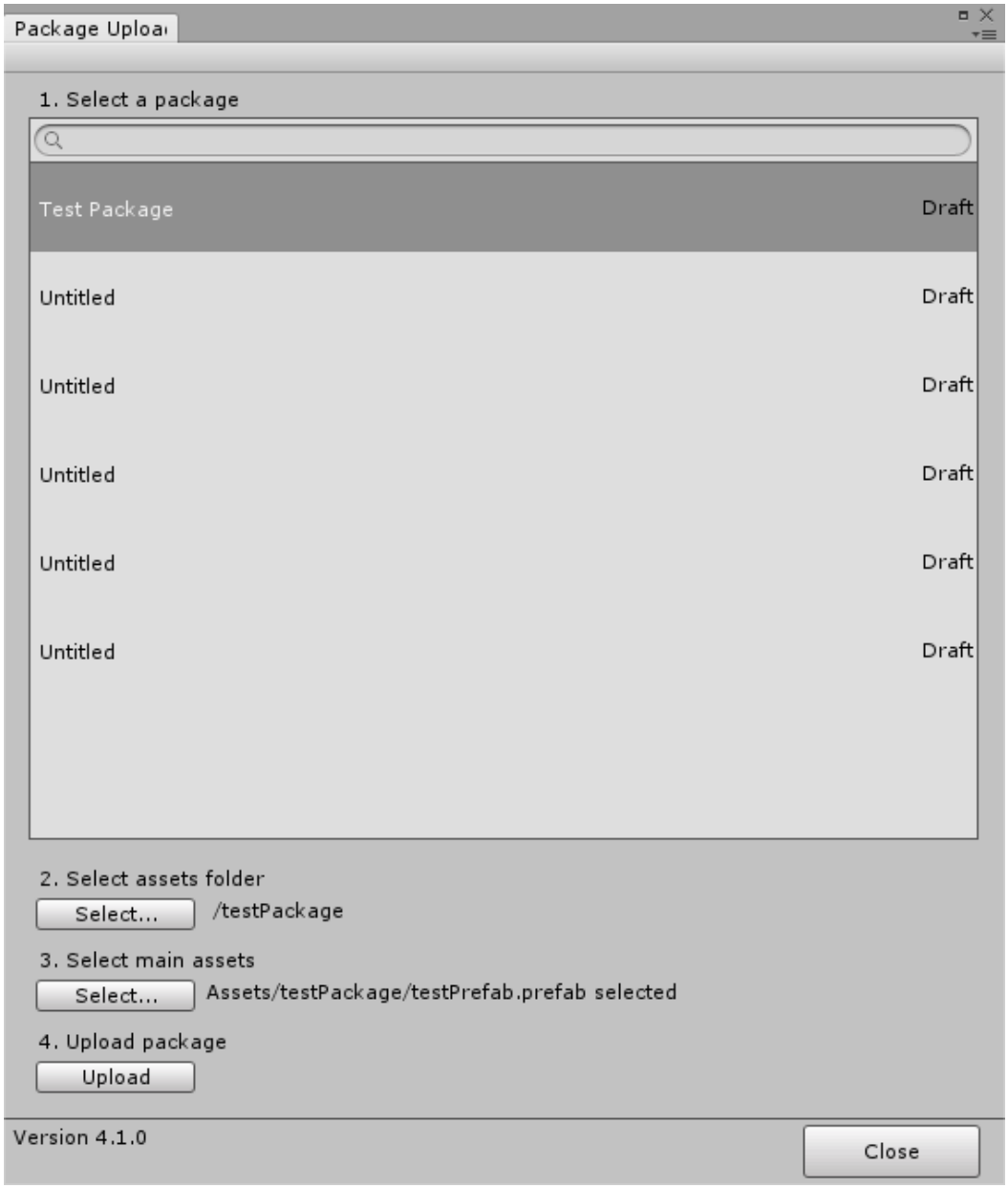
In the Unity Editor, import the Asset Store Tools into your Project.

Go to **Asset Store Tools > Package Upload**. This dialog box appears:



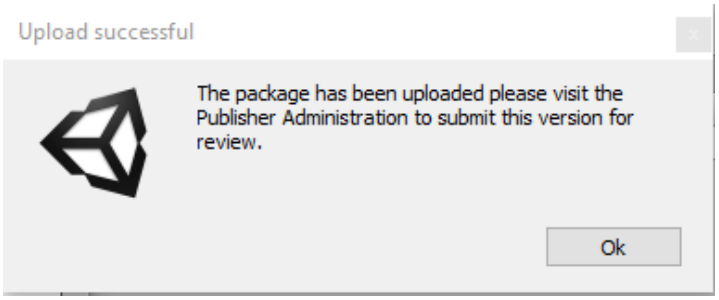
Enter your Unity Publisher account information into this box. The Package Upload tool fetches your drafts and packages via your Publisher Portal. When the tool has fetched your packages from the Portal, the Package Upload window appears.

Select the folder that contains your entire Project, and then select an Asset to showcase within that Project. If you have multiple draft packages in your publisher account, make sure you click on the package for the Asset you are submitting. The Package Upload window shows the folder and Assets you are uploading:

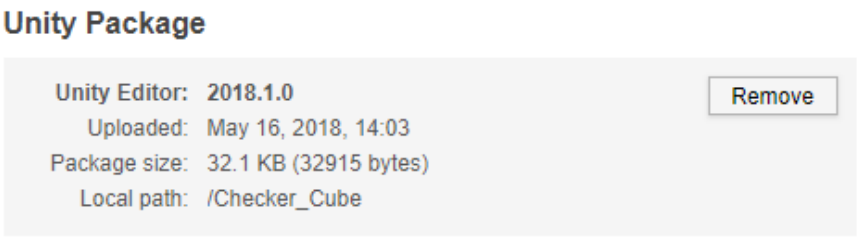


Click **Upload** to connect your Assets to your Publisher Portal. The status bar next to the title of your selected folder shows the progress of the upload.

When upload is complete, the Editor displays a dialog box to verify that the package is now on your Publisher Portal:



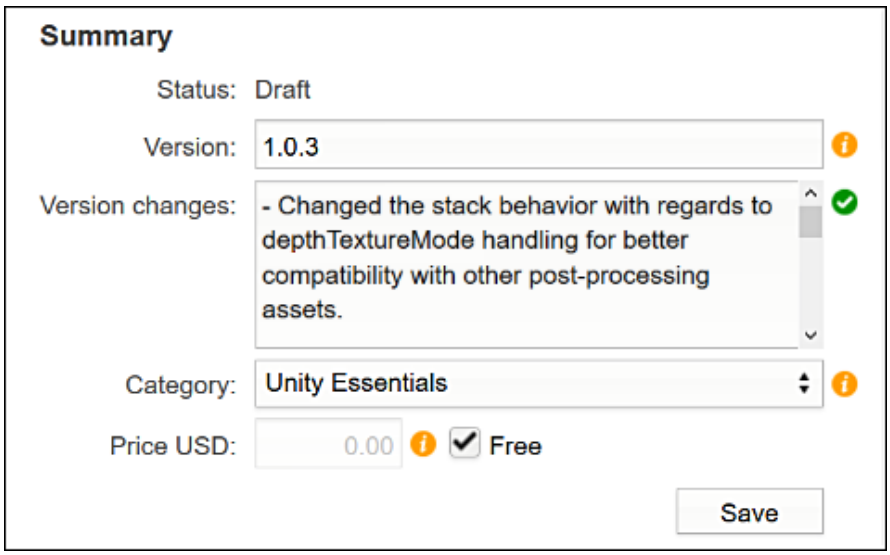
Your package is now available on the Publisher Portal:



Continue to prepare your package submission for review on the Publisher Portal.

Summary

In the summary of your asset you can change things like the ***Version Number*** and the ***Price USD*** of your asset.



Status: Current Status of the asset. (*Draft, Published, Declined, Pending*)
Version: The version number for your asset (*eg; 1.5, 1.6, 1.7*)
Version Changes: A place to enter notes on what you have changed to the asset (*mainly used for updates*)
Price USD: The price you wish for customers to buy your asset for.

Naming your asset and adding a description

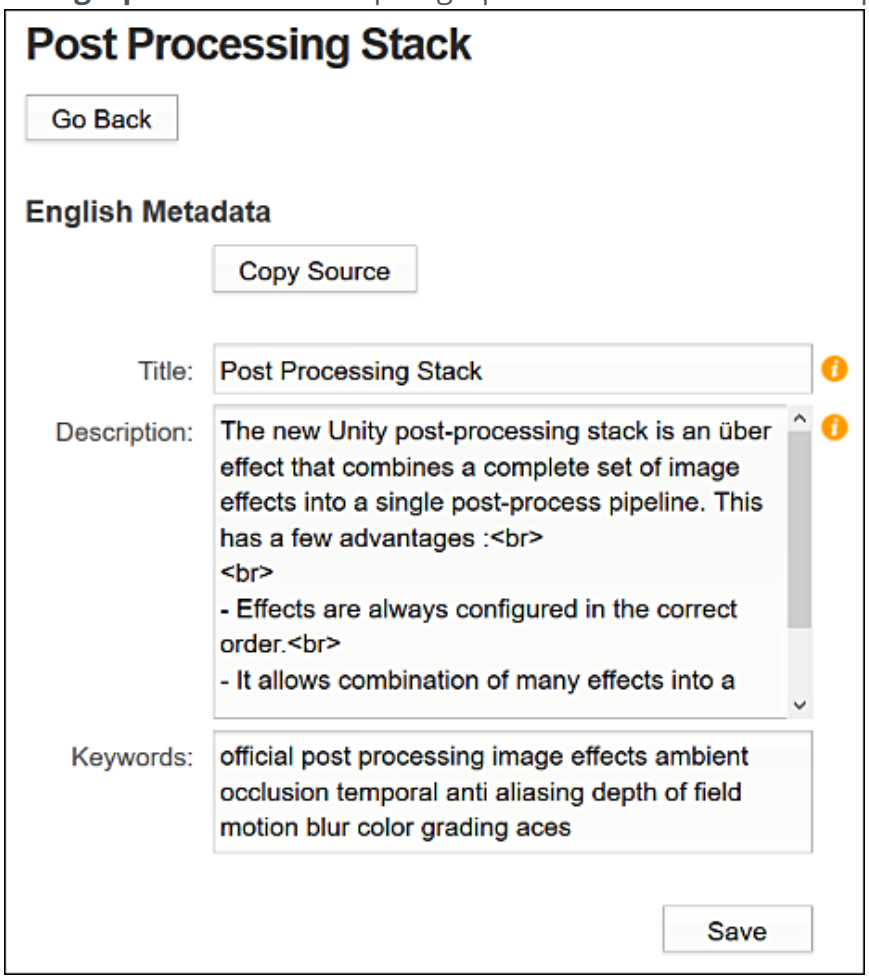
You can change the name of your asset by clicking the Edit link below the *"Metadata"* tab under **Metadata and Artwork**.

Title: This is the name of your asset. *This field is required*
Description: This is the description of your asset (*Description may only contain the EM, STRONG, A, P and BR tags*)

Tagging your Description:

The description section of your asset requires tags in order to use bold text or break to a new line. Here is a table below to help you.

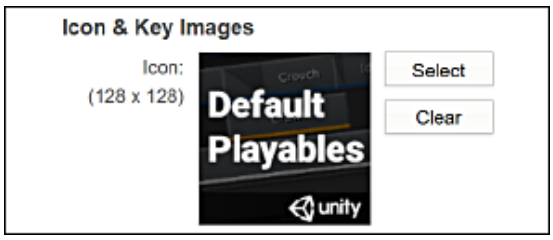
Bold:	This allows you to make the containing text bold	Bold Text
Italic:	This allows you to make the containing text italic	<e>italic text
Link:	This allows you to add a URL link	My Link
Break:	This allows you to make a line break	
Paragraph:	This Defines a paragraph.	A Paragraph



Adding Key Images

You can change the Key Images of your asset by clicking the Edit link below the ***“Key Images”*** tab under **Metadata & Artwork**.

Upload and Remove Image: Click on the ***Select*** button to upload the image and Click ***Clear*** to remove it.



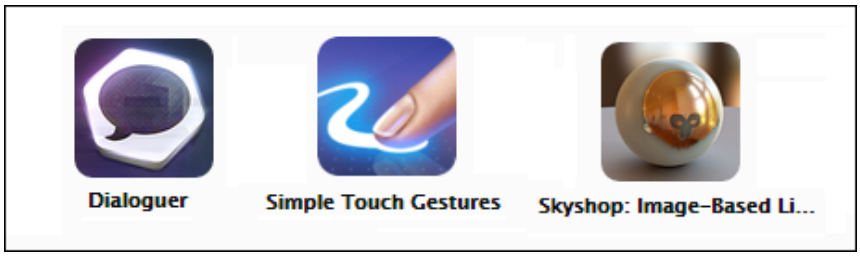
Icon & Key Image Sizes:

Icon: 128 x 128 (*pixels*)

Small: 200 x 258 (*pixels*)
Large: 860 x 389 (*pixels*)
Eye Catching: These key images must aim to capture the eyes of customers just as a colorful cartoon captures the eyes of children. A lot of customers browse on the asset store and can commonly buy under impulse. Your Key images must help your asset stand out.

Icon Images:

The Icon Image is the most common key image that users will see. Here are a few well made icon images from the asset store as examples:



Adding Screenshots

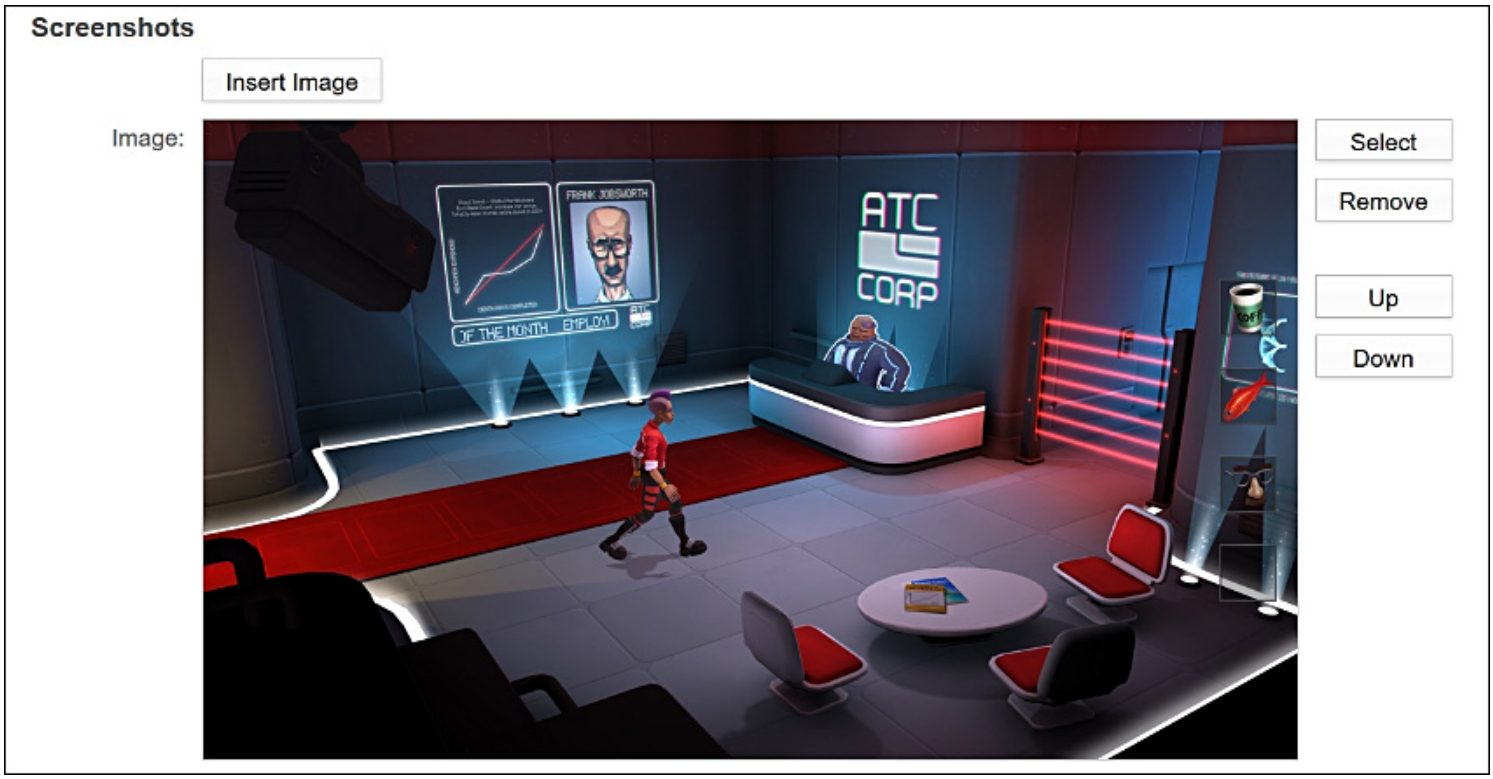
You can change the screenshots attached to your asset by clicking the Edit link below the ***"Screenshots"*** tab under **Metadata & Artwork**.

Click the **"Insert Image"** button in order to upload the image and add it as a screenshot for your asset. You can add multiple screenshots this way. Remove a screenshot by clicking the **"Remove"** button next to the image you wish to remove.

The **"Up"** and **"Down"** buttons allow you to put your screenshots into any order you like.

How screenshots :

Screenshots are just as important as key images when trying to sell your asset. Your asset may be the best asset in the world, however, if your screenshots do not communicate that then it will not sell.



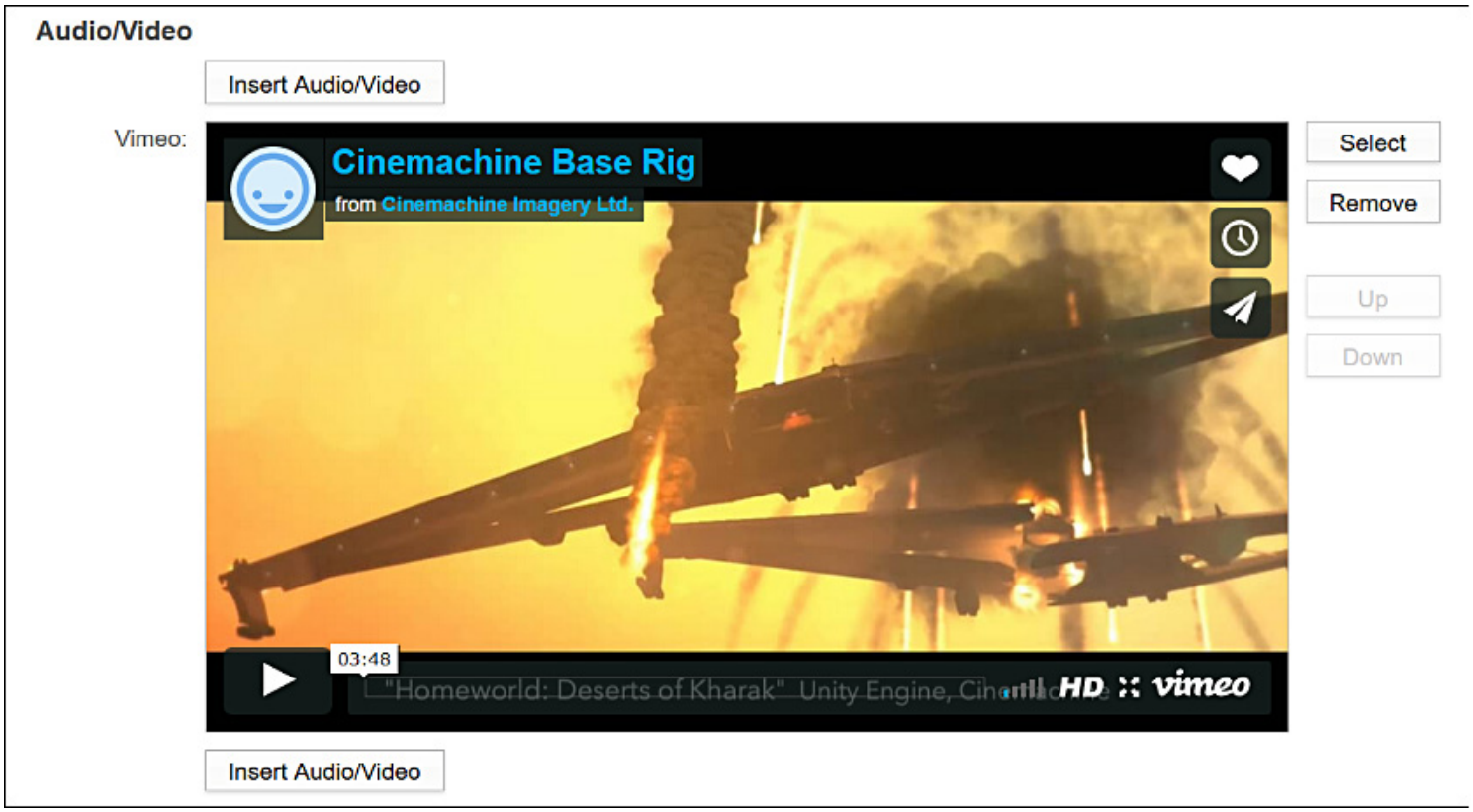
Adding screenshots and making them help sell your asset

Adding Audio and Video

You can change the name of your asset by clicking the Edit link below the ***"Audio/Video"*** tab under **Metadata & Artwork**.

This allows you to add Audio or Video to the asset and embed it in the show reel with the screenshots of your asset. This is done by providing a link to either of the services below:

YouTube
Vimeo
SoundCloud
MixCloud
SketchFab
Adding Audio/Video to your asset



Deleting a Draft

You can delete a draft by clicking the drop down text field under the Packages tab of your Publisher Administration page and selecting draft. This will deisplay your drafts.

Click on the name of the draft you wish to delete and click on the ***"Delete Draft"*** button below the **Metadata & Artwork** section at the bottom of the page.

2018–06–29 Page published with limited editorial review

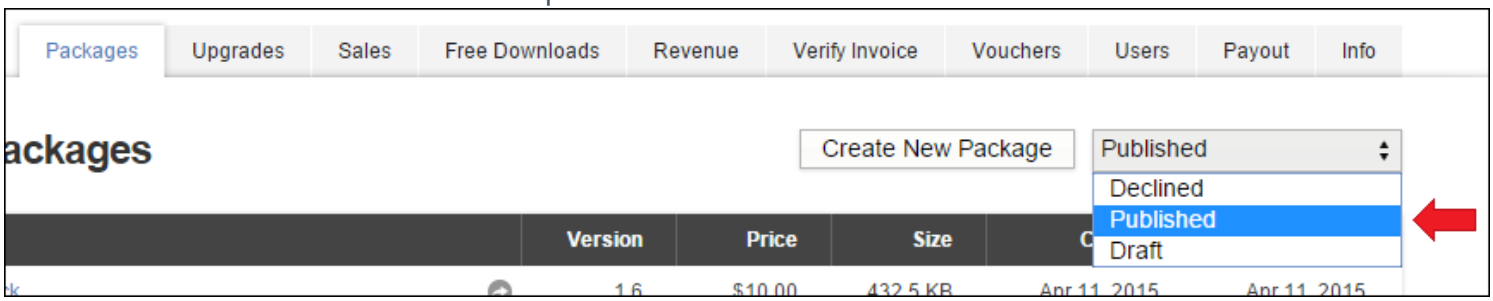
DeprecateAssetGuide

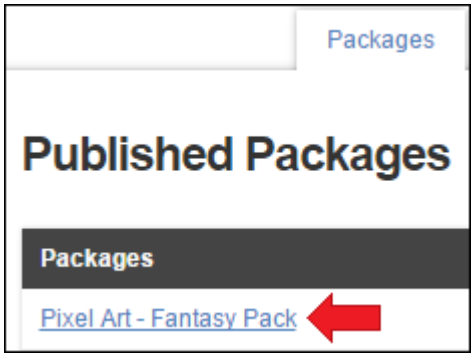
Leave feedback

When a package is deprecated it is no longer available in the **Asset Store**. It is removed from the store. Customers that have already purchased or downloaded the package will still be able to download the package. To get a deprecated package back in the store, a new draft needs to be submitted and reviewed.

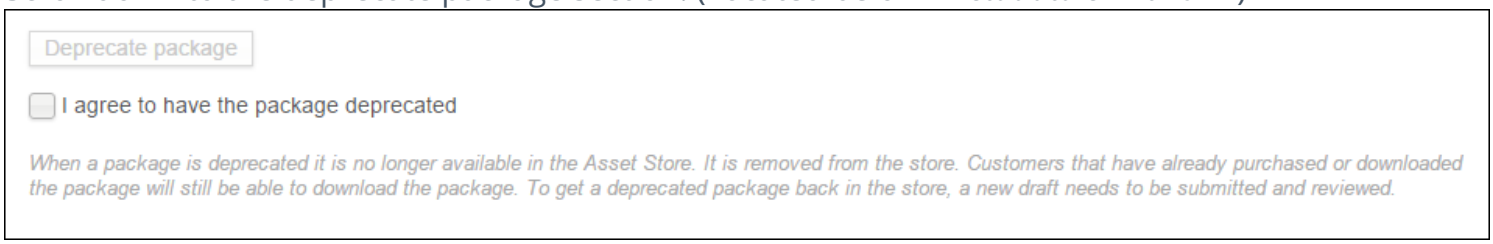
In this guide we are going to depricate the **asset** "**Pixel** Art - Fantasy Pack". The steps will be the same for any **asset** that is currently published to the **Asset** Store.

Log-in to Publisher Administration
Go To the Packages Tab.
Select the **Published** view in the drop down menu.

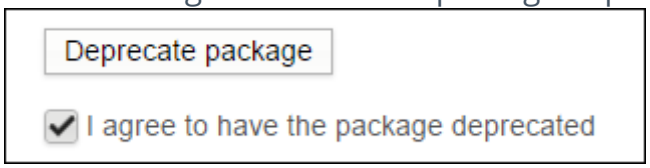




Click the name of the package you wish to deprecate.
Scroll down to the deprecate package section. (Located below *"Metadata & Artwork"*)



Click the "I agree to have the package deprecated" check box.



Click the "Deprecate Package" button.

Asset Store Manual

Leave feedback

This is the table of contents for all **asset store** related subjects for both customers and publishers. This page contains guides; from downloading your **assets** from the **asset store** to publishing your own **assets**.

Customers

This section describes how to accomplish some common tasks with the Asset Store.

Publishers

This section describes how to accomplish some common tasks with the Asset Store Tools and the Publisher Administration section of your web account.

Publisher Admin Section Overview

Viewing the status of your Asset store submissions

Refunding Your Customers

Removing your Assets

Providing support on your assets

How to Promote your Assets

Publisher Admin Section Overview

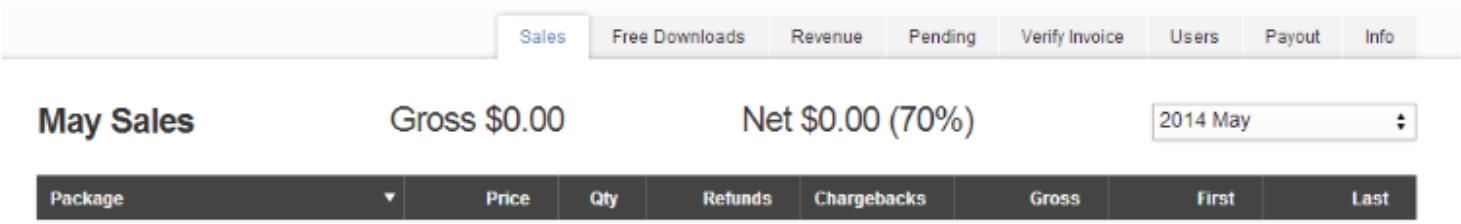
Leave feedback

The Publisher Administration section is a place where you can manage your presence as a publisher. This section will tell you about each tab within the publisher admin section and how it can help you.

Packages

The packages tab is where you manage your **assets** and create new packages for submission. See *Asset Store Publishing Guide* for more help on Submitting to the store and *Updating* ***Asset Packages*** for more help on managing and updating your existing assets.

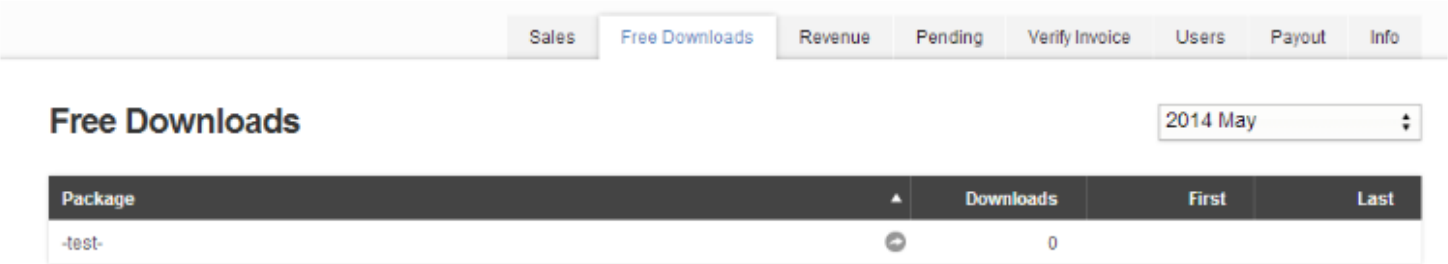
Sales



The Sales tab allows you to view the sales data on all your assets It will show you the Gross amount and your Net amount for the selected month. You can view the sales figures for different months by changing the month from the drop down box.

Package: Name of the Asset. *includes a button that directs you to your asset*
Price: The cost of the product at the time the information on that row was recorded.
Qty: The Quantity of purchases on the asset in that row.
Refunds: The number of Refunds completed on the asset in that row.
Chargebacks: The number of Refunds completed on the asset in that row. A chargeback is a demand by a credit-card provider for a retailer to make good the loss on a fraudulent or disputed transaction.
Gross: The gross amount of money taken for the sales on the asset on that row.
First: The date of the first purchase of the asset on that row for the month selected.
Last: The date of the last purchase of the asset on that row for the month selected.

Free Downloads

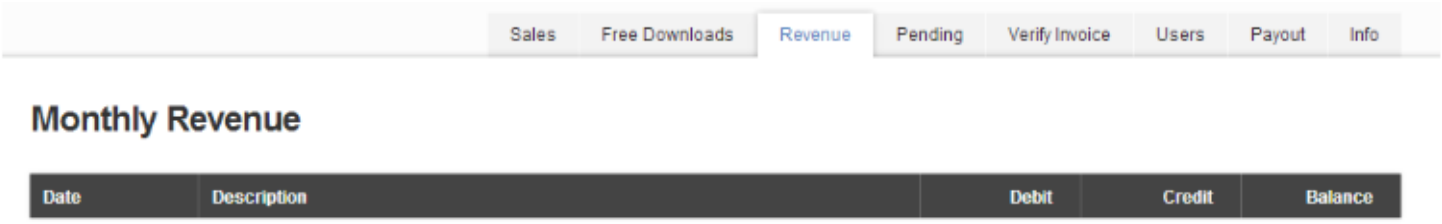


This page shows you your free assets. It shows you the following:

The names of all the packages you have ever published for free.
The number of downloads on that package.

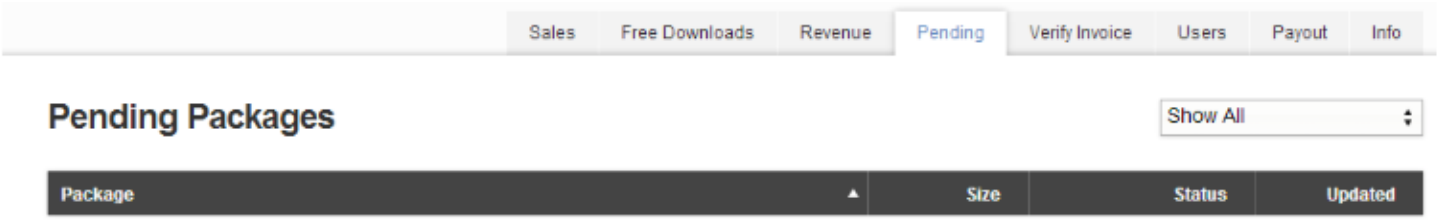
The date of the first download.
The date of the last download.

Revenue:



The Revenue tab is a place where you can view your revenue earned for each month and also see when your earnings are paid out to your bank or PayPal.

Pending:



The Pending tab allows you to view the status of submitted assets. The table will show you the following:

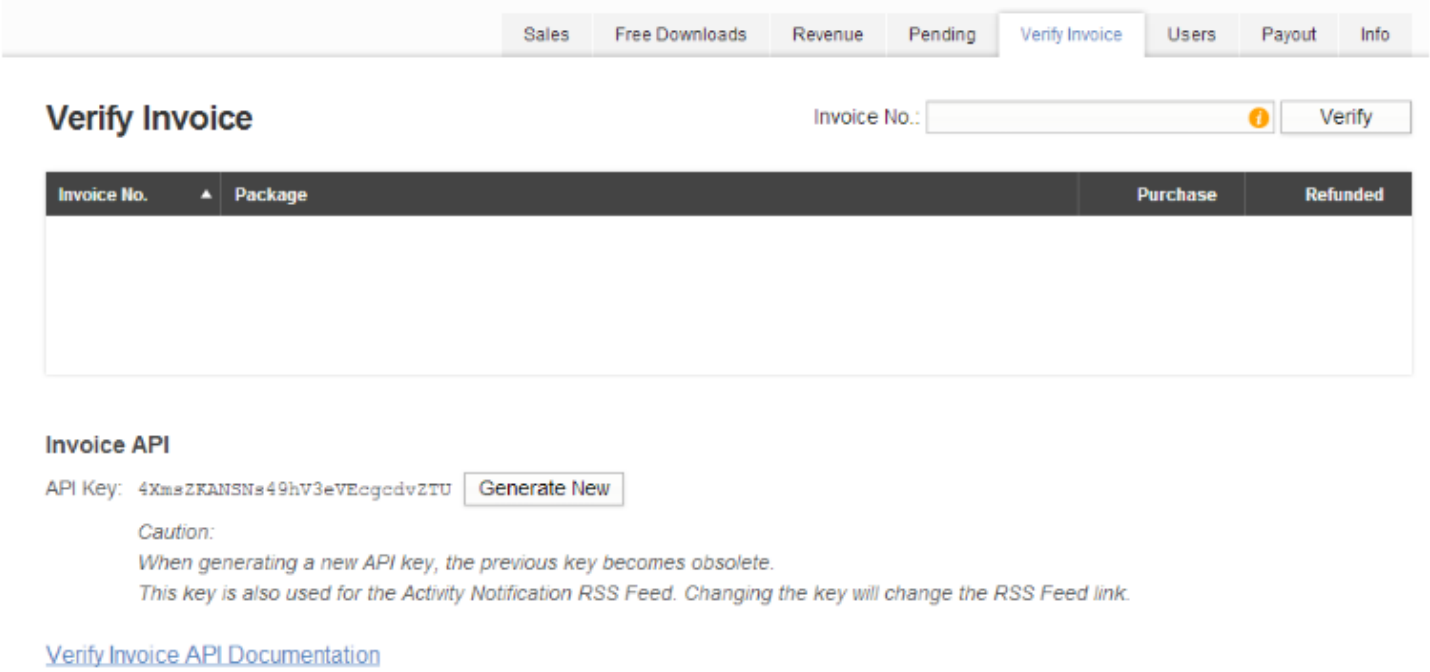
Package: Displays the name given to the submitted asset.

Size: Total size of the submitted package.

Status: The status of the asset. Here is an overview of them:

Pending Review: This means the asset has been received and is in the submission queue and is awaiting review from the vetting team. Only contact assetstore@unity3d.com if the asset has not been reviewed within 10 working days of submission.
Uploaded: If an asset is displayed as Uploaded for more than 2 hours then the asset will need resubmitting.
Draft: This will mean that the asset has not been submitted and is currently in pre submission stage.

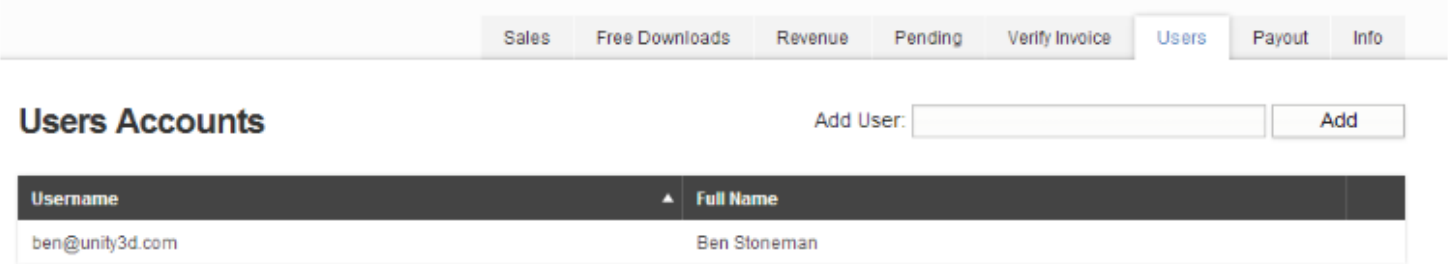
Verify Invoice:



The **Verify Invoice** tab allows you to enter the invoice number of an **asset store** customer to see if the invoice is valid.

It will also display your assets if they are present on the invoice and if they have already been refunded.

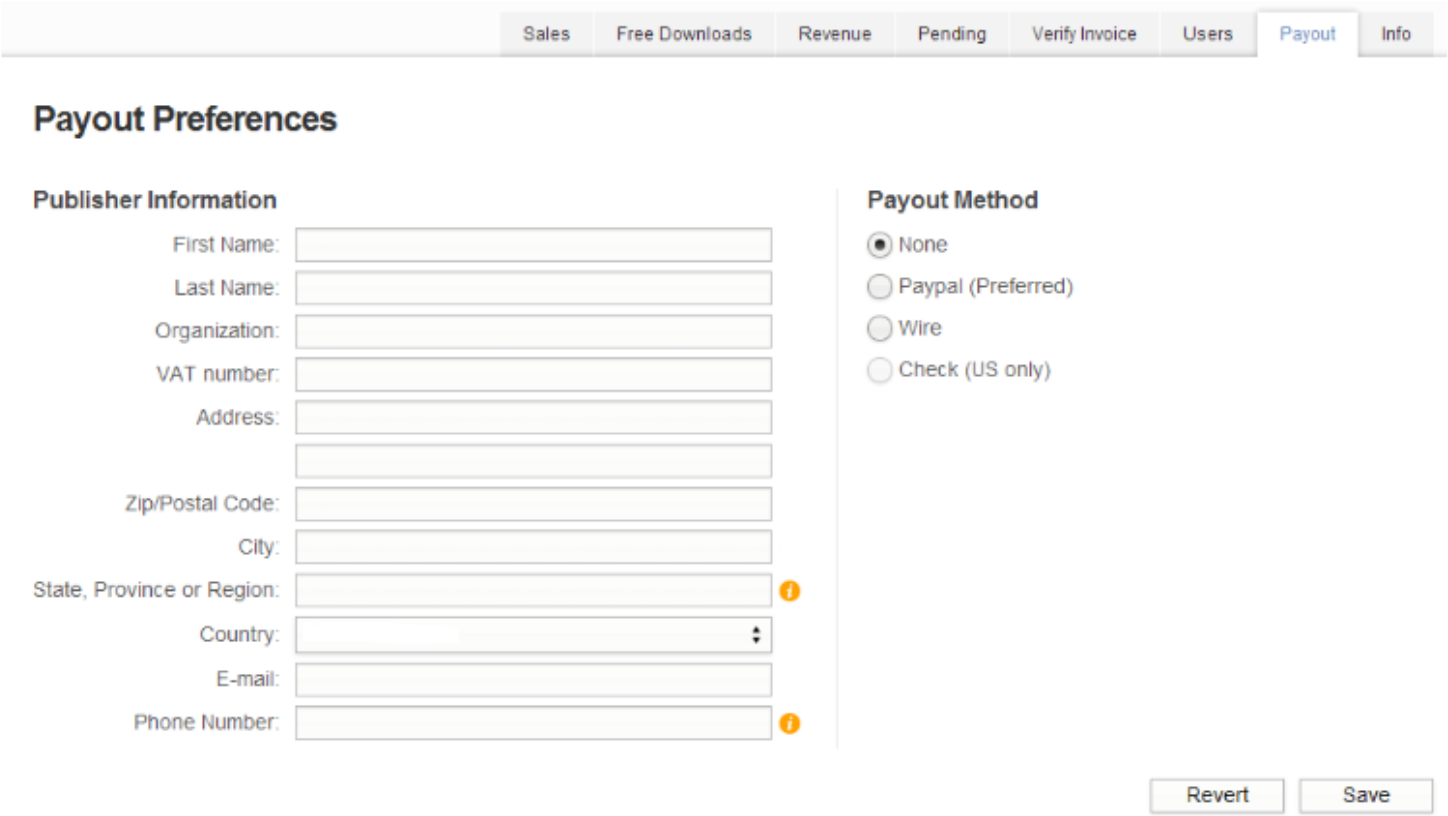
Users:



The Users tab allows you to view and add Unity accounts to the publisher admin section. This means that if you have a team, each person can be added if needed.

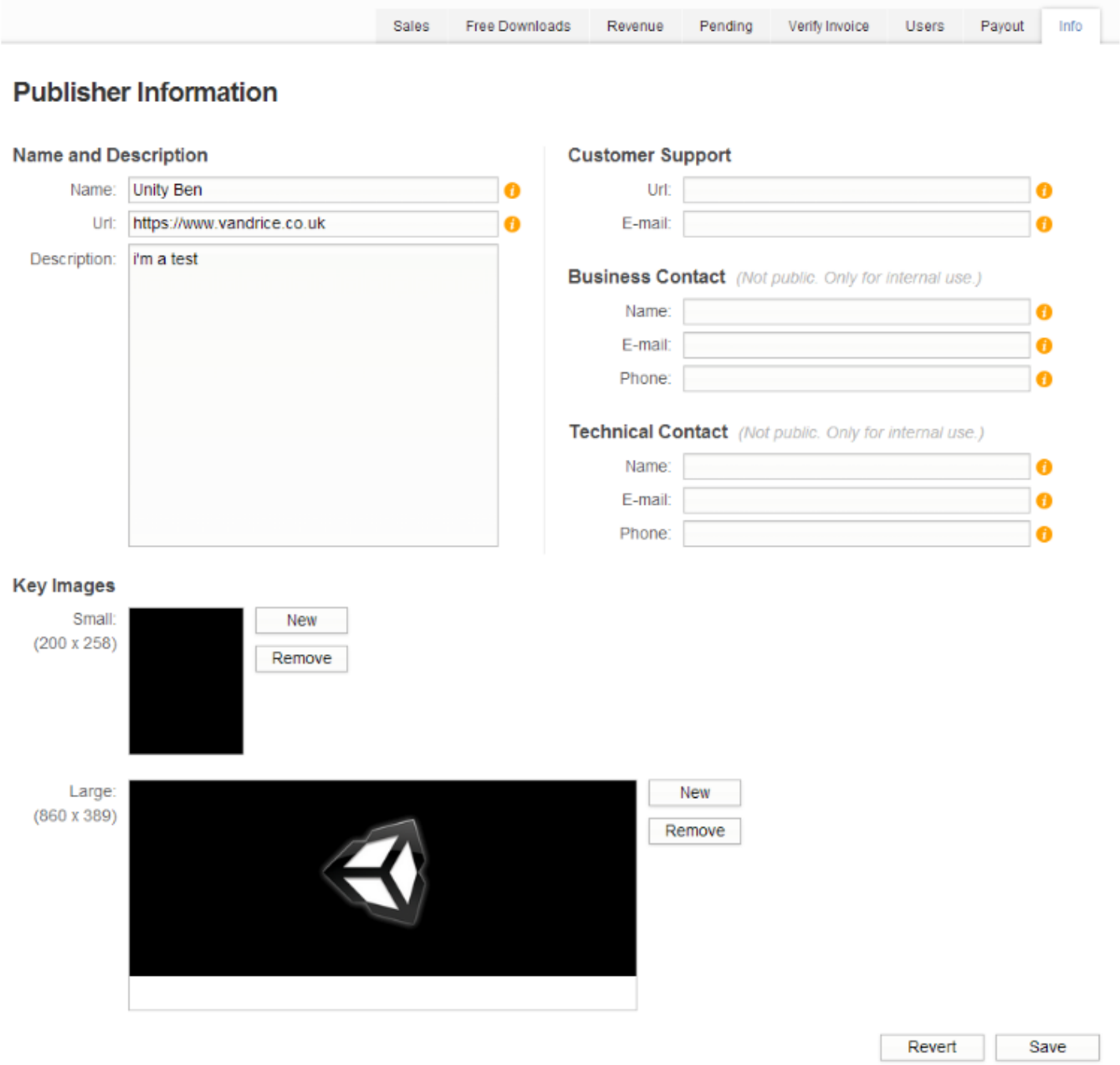
When you request a refund on behalf of the customer support will check the users to confirm the request has come for a valid e-mail attached to the respective publisher account.

Payout:



The Payout tab allows you to manage the publisher information and payout options for the revenue earned by that publisher account.

Info:



The Info tab allows you to edit the profile of the publisher account. The page is very much self explanatory, however, here is a brief overview of the sections:

Name and Description: For editing your publisher name, description and main webpage.

Customer Support: Allows you to provide a support webpage and/or support e-mail address.

Business Contact: This is used to provide contact information regarding business. This is used for internal use.

Technical Contact: This is used to provide contact information regarding the technical side. This is used for internal use.

Key Images: The key images that are displayed on your publisher page. The small key image is not currently used.

Setting up Google Analytics

Leave feedback

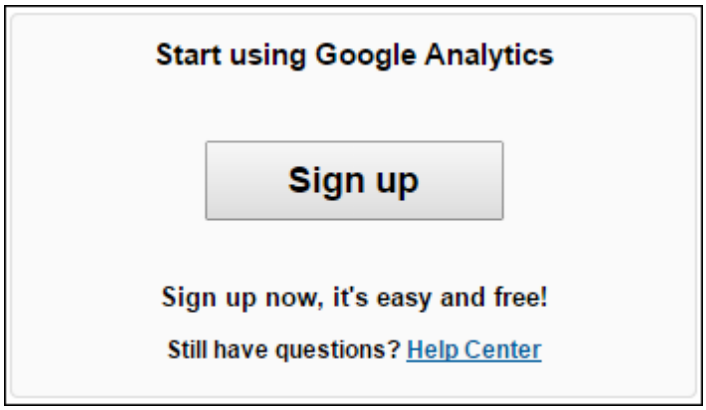
What is Google Analytics?

Google **Analytics** is a web-based information center that displays statistics on websites and web pages. As a publisher, you may have seen the **Tracking ID** text field under the info tab of your **Asset Store Publisher Administration** section.

How do I set it up?

1. Sign in or sign up

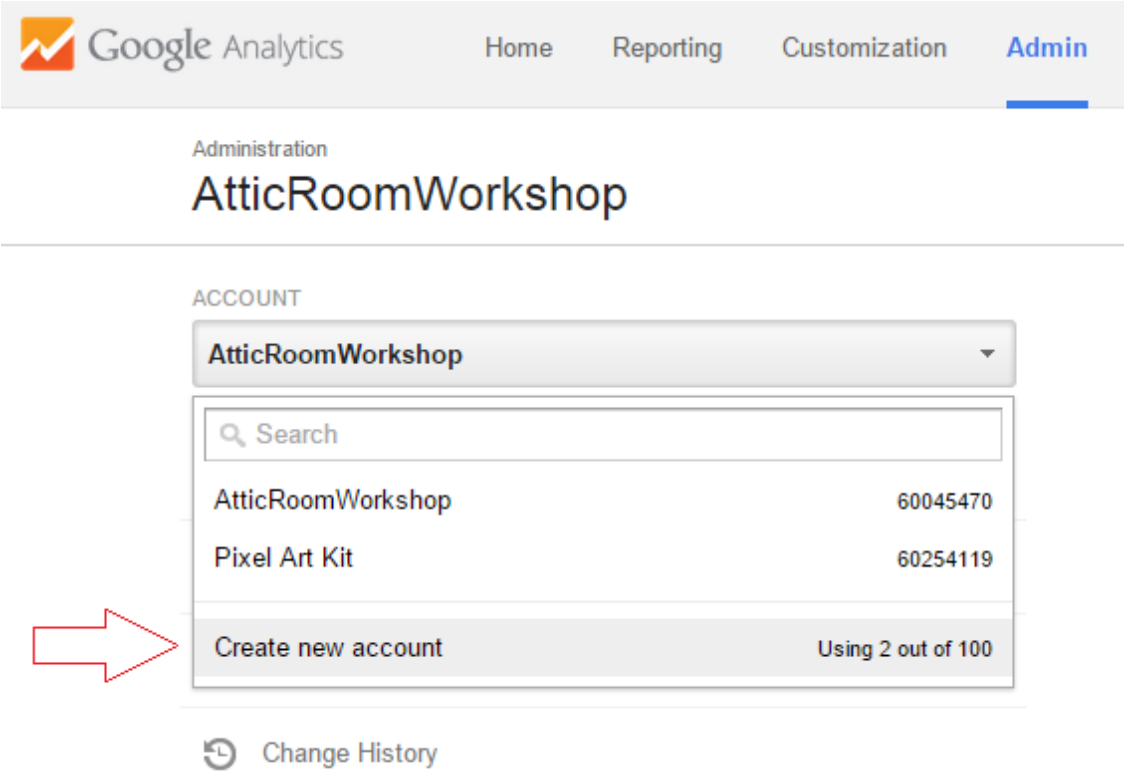
Go to the Google Analytics home page, and sign in with your Gmail account.



2. Create a new Google Analytics account

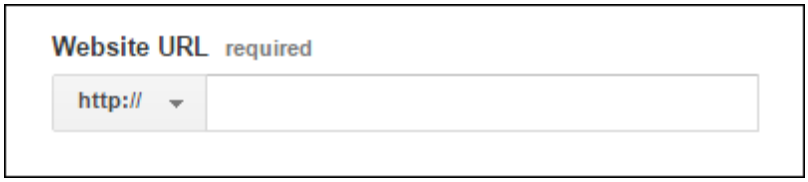
In Google Analytics, a “new account” refers to a new link or URL that you want to track. If this is the first time you have logged into Google Analytics, the site prompts you to do this automatically.

To create a new account, go to **Admin** > **Accounts** > **New Account**.

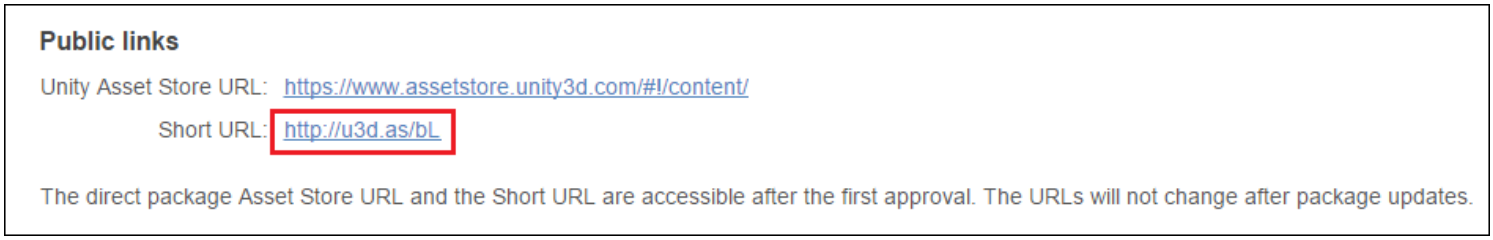


3. New account form

The Google Analytics new account form asks you to add a website URL.

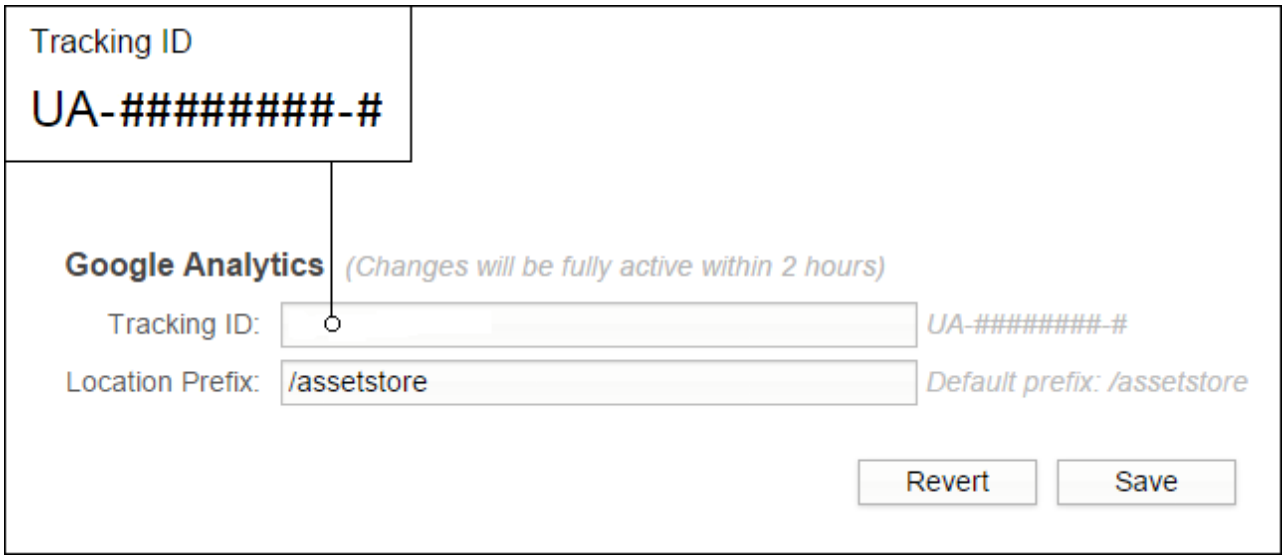


Here, you need to enter the Short URL for your **asset**. To find the Short URL, click the name of your **asset** in the **Packages** tab of the **Asset** Store Publisher Administration section and scroll to the bottom of the page:



4. Tracking ID

Creating a new account generates a Tracking ID. Copy the Tracking ID and paste it into the **Tracking ID** text field at the bottom of the **Info** tab in the Asset Store Publisher Administration section. For the **Loction Prefix**, you can choose to leave the default "/assetstore".


Tracking ID
UA-########-#
Google Analytics
(Changes will be fully active within 2 hours)
Tracking ID:
UA-########-#
Location Prefix:
/assetstore
Default prefix: /assetstore
Revert
Save

Viewing the status of your Asset Store submissions

Leave feedback

You can view the publishing status of your **asset** via the **Pending** tab in your Publisher Administration Page

Submission status

Pending:	This shows all assets that have been received and are in the submission queue awaiting review from the vetting team. You should only contact assetstore@unity3d.com if the asset has not been reviewed within 14 working days of submission.
Published:	This will show all assets that are currently published on the **Asset Store**.
Declined:	This shows all assets that were submitted and declined by the Asset Store vetting team.
Disabled:	This will show all assets that have been disabled from the Asset Store.

If the asset does not appear in the pending queue, then it has either been been accepted to the store, or it has not been created as a package.

If your asset is removed from the store for any reason, a notification email is sent to the main email address attached to your publisher account.

Refunding your customers

Leave feedback

Customers may contact you to request a refund on your **asset**. However, as the purchase was made through the **Asset Store**, it should be the store that manages the refund - not individual publishers.

If possible, try to help your customer resolve their issue to avoid a refund. If a refund is required, then you must request it form the Asset Store on the customer's behalf, using the following process:

Ask the user to provide you with the invoice they received when they purchased the asset.

Log in to your publisher administration page.

Verify the invoice using the **Verify Invoice** tab.

Once verified, email assetstore@unity3d.com from an email address attached to your publisher account. You must provide either the invoice as an attachment, or the invoice number.

Here is a template request:

```
Hi Asset Store Team,

Please refund the following asset(s)

Publisher Name:

- Asset name(s)

Regards
```

Removing your Assets from the Asset Store

Leave feedback

To disable an **assets** that you have successfully published to the **Asset Store**, email assetstore@unity3d.com with the following:

Publisher name
Asset name
Confirmation that you want to disable the asset(s) stated

The request must be sent from the email address attached to the publisher account of the asset(s) you want to disable.

Providing support

Leave feedback

As an **Asset Store** publisher, you are required to provide customer support on all your published **assets**.

You need to ensure that you have an email address for users to contact you. This should be entered into the **Info** tab of your **Publisher Administration** account.

We ask that you handle support requests from users in a professional manner. If we recieve complaints about a publisher's conduct in handling customer support requests, we may review the suitability of that publisher for the Unity Asset Store. This is also stated in your Asset Store provider agreement.

Asset Store promotions

Leave feedback

If you would like your **Asset** package to be included in any of Unity's promotional sales, please apply to join a sale via the Asset Store Sales Application Form page.

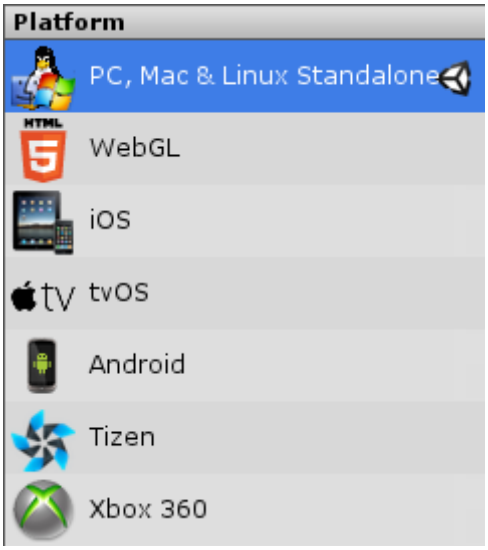
Our sales candidates are chosen at least a month or two ahead of the sales. As such, it may be some time before your Asset is included. Please note that submitting a Sales Application Form does not guarantee that your packages will be included in a sale; **Asset packages** are chosen based on which ones we feel will market well, perform well in the Store, or that best fit the specific theme of a promotion.

Rather than taking part in a Unity promotional sale, you might also want to change the price of your Asset for your own promotion. If you do this, the promotion must be advertised through your own means (on your own website or social media page). Details of this sale must be provided in the description text for your Asset while the sale is ongoing, and this must include the regular price and the length of the promotional sale period.

Unity does not allow any sale-related graphics throughout your Asset package's marketing and sales imagery. You may not promote your product as being included in any sales or features hosted by Unity Technologies that you are not actually included in. If we notice any Asset package that doesn't repect these guidelines, we reserve the right to return the Asset package to its former regular price without notice.

Platform-specific

Leave feedback



This section covers specific information for the many non-desktop and web platforms supported by Unity.

Note that many platforms such as PlayStation, Xbox and Wii only make their platform-specific documentation available locally on your machine after installation. To find this, open the Unity Editor and navigate to **Help > Unity Manual**.

Related tutorials: Mobile & Touch

See the Knowledge Base Editor section for troubleshooting, tips and tricks.

Standalone

Leave feedback

This section of the User Manual contains documentation on developing for standalone platforms. **Standalone** refers to builds for Mac, Windows and Linux.

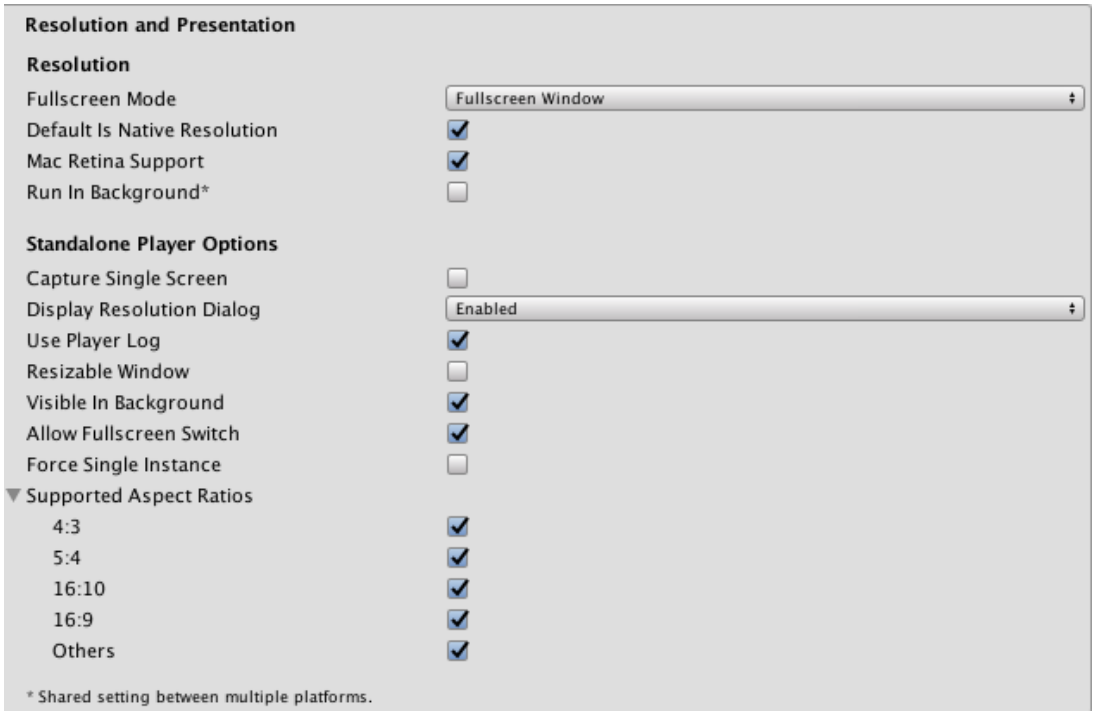
Use the navigation column on the left-hand side of the page to view topics in this section.

Standalone Player Settings

Leave feedback

This page details the **Player Settings** specific to standalone platforms (Mac OSX, Windows and Linux). A description of the general Player Settings can be found here.

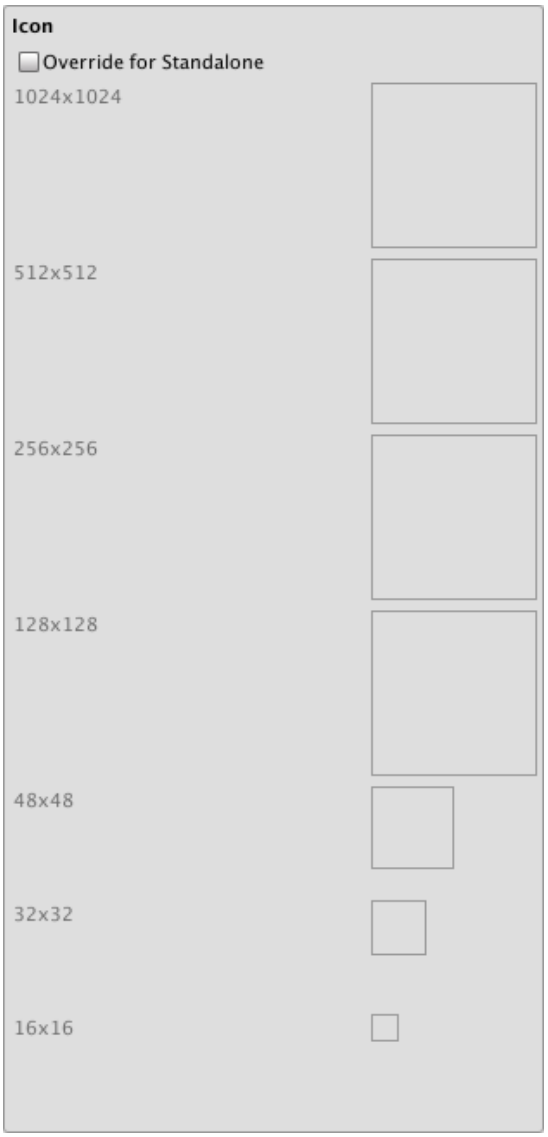
Resolution And Presentation



Property:	*Function:*
Resolution	
Fullscreen Mode	Choose the full-screen mode. This defines the default window mode at startup.
Fullscreen Window	The Fullscreen Window mode sets your app window to the full-screen native resolution of the display. Unity renders app content at the resolution set by script (or by user selection when the built application launches), but scales it to fill the window. When scaling, Unity adds black bars to the rendered output to match the **aspect ratio** chosen in the Player Settings, so that the content isn't stretched. This process is called letterboxing.
Exclusive Fullscreen	**Exclusive Fullscreen** mode sets your app to maintain sole full-screen use of a display. Unlike **Fullscreen Window**, this mode changes the OS resolution of the display to match the app's chosen resolution. **Exclusive Fullscreen** is only supported on Windows; on other platforms, the setting falls back to **Fullscreen Window**.
Maximized Window	**Maximized Window** mode sets the app window to the operating system's definition of "maximized". On macOS, this means a full-screen window with an auto-hidden menu bar and dock. **Maximized Window** is only supported on macOS; on other platforms, the setting falls back to **Fullscreen Window**.
Windowed	Windowed mode sets your app to a standard, non-full-screen, movable window, the size of which is dependent on the app resolution. In Windowed mode, the window is resizable by default. To disable this, disable the Player Settings **Resizable Window** setting.
Default Is Native Resolution	Check this box to make the game use the default resolution used on the target machine.
Default Screen Width	Default width of the game screen in **pixels**.

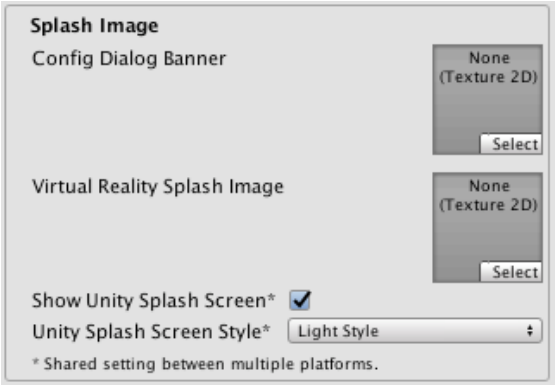
Property:	*Function:*
Default Screen Height	Default height of the game screen in pixels.
Mac Retina Support	Check this box to enable support for high DPI (Retina) screens on a Mac. Unity enables this by default. This enhances Projects on a Retina display, but it is somewhat resource-intensive when active.
Run in background	Check this box to make the game keep running (rather than pausing) if the app loses focus.
Standalone Player Options	
Capture Single Screen	Check this box to ensure standalone games in fullscreen mode do not darken the secondary monitor in multi-monitor setups. This is not supported on Mac OS X.
Display Resolution Dialog	Choose whether the game should start with a dialog to let the user choose the screen resolution. The options are **Disabled**, **Enabled** and **Hidden by Default** (i.e. the option only appears if the alt key is held down at startup).
Use Player Log	Check this box to write a log file with debugging information. If you plan to submit your application to the Mac App Store, leave this option un-ticked. Ticked is the default.
Resizable Window	Check this box to allow the user to resize the standalone player window.
Visible in Background	Check this box to show the application in the background if Fullscreen Windowed mode is used (in Windows).
Allow Fullscreen Switch	Check this box to allow default OS full-screen key presses to toggle between full-screen and windowed modes.
Force Single Instance	Check this box to restrict standalone players to a single concurrent running instance.
Supported Aspect Ratios	Choose the aspect ratios that appear in the Resolution Dialog at startup (as long as they are supported by the user's monitor).

Icon



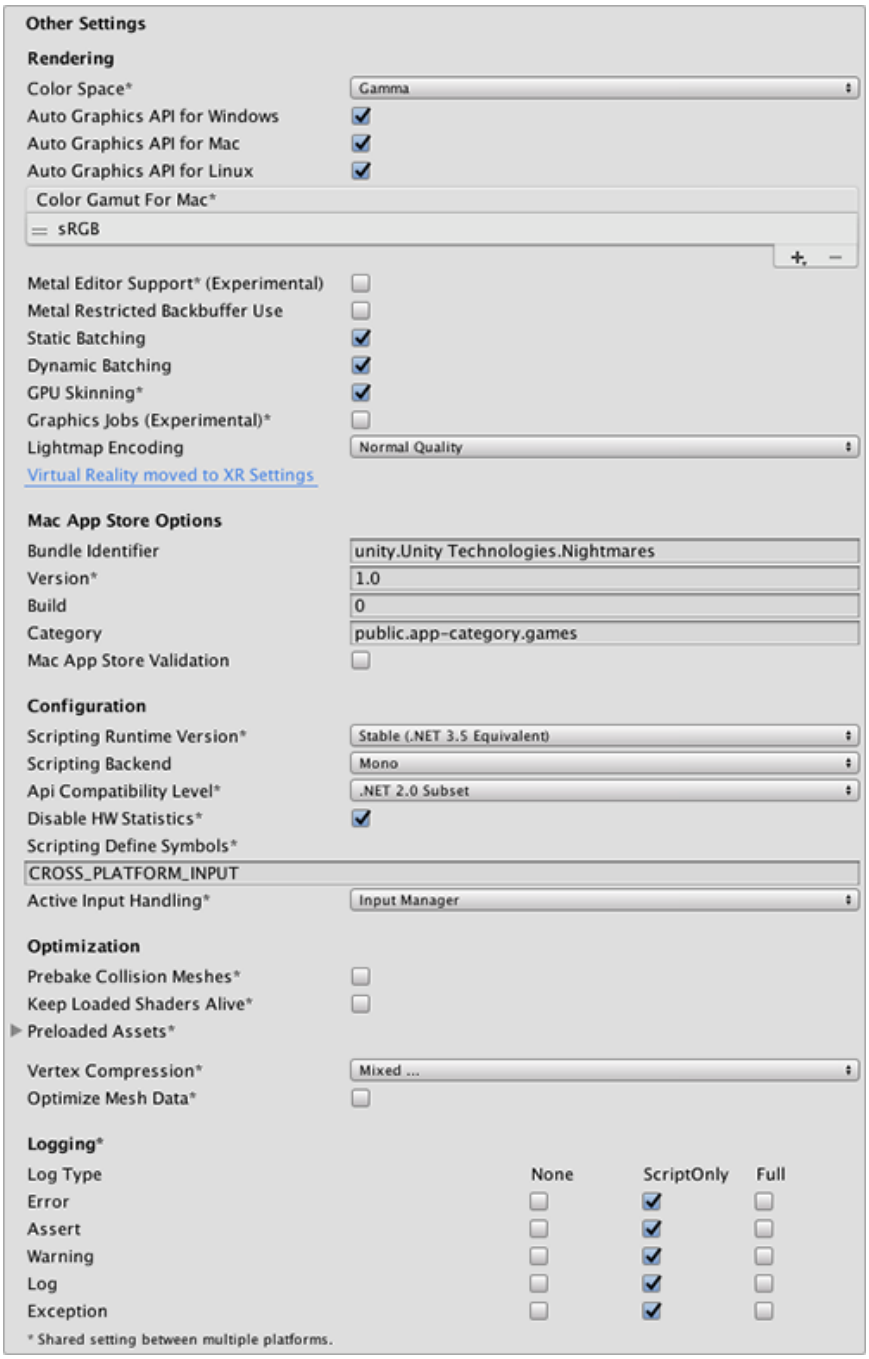
Property:	*Function:*
Override for Standalone	Check this box to assign a custom icon to be used for your standalone game. Upload different sizes of the icon to fit each of the squares below the checkbox.

Splash Image



Property:	*Function:*
Config Dialog Banner	Add a custom splash image to be displayed in the **Display Resolution Dialog**.
Show Unity Splash Screen	Shows the “Made with Unity” Splash Screen when the game is loading .

Other Settings



Property:	*Function:*
Rendering	
Color Space	Choose which color space should be used for rendering. The options are **Gamma** and **Linear**. See the Unity Manual page on Linear Rendering for a guide to the difference between the two.
Auto Graphics API for Windows	Check this box to use the best Graphics API on the Windows machine the game is running on. Uncheck it to add and remove supported Graphics APIs.
Auto Graphics API for Mac	Check this box to use the best Graphics API on the Mac the game is running on. Uncheck it to add and remove supported Graphics APIs.
Auto Graphics API for Linux	Check this box to use the best Graphics API on the Linux machine it runs on. Uncheck it to add and remove supported Graphics APIs.
Metal Editor Support (Experimental)	Makes the Unity Editor use the Metal API and unlocks faster **shader** iteration for targeting the Metal API.

Property:	*Function:*
Metal Restricted Backbuffer Use	Allow improved performance in non-default device orientation. This sets the frameBufferOnly flag on the back buffer, which prevents readback from the back buffer but enables some driver optimization.
Static Batching	Check this box to use Static batching.
Dynamic Batching	Check this box to use Dynamic Batching (activated by default).
GPU Skinning	Check this box to enable DX11/ES3 GPU **skinning**.
Graphics Jobs (Experimental)	Check this box to instruct Unity to offload graphics tasks (render loops) to worker threads running on other CPU cores. This is intended to reduce the time spent in `Camera.Render` on the main thread, which is often a bottleneck. **Please note that this feature is experimental**. it may not deliver a performance improvement for your project, and may introduce new crashes.
Lightmap Encoding	Affects the encoding scheme and **compression** format of the **lightmaps**. Choose from Normal Quality and High Quality.
Mac App Store Options Configuration	See Delivering your application to the Mac App Store.
Scripting Runtime Version	Choose which .NET runtime to use in your project. For more details, see Microsoft's .NET documentation.
.NET 3.5 Equivalent	A .NET runtime which implements the .NET 3.5 API. This functionality is deprecated, and should no longer be used. Please use .NET 4.
.NET 4.x Equivalent	A .NET runtime which implements the .NET 4 API. This API is newer than .NET 3.5, and as such, it offers access to more APIs, is compatible with more external libraries, and supports C# 6. This is the default scripting runtime.
Scripting Backend	**Mono2x** is the only scripting backend supported on Standalone platforms.
API Compatibility Level	There are two options for API compatibility level: **.Net 2.0**, or **.Net 2.0 Subset**.
Disable HW Statistics	Check this box to instruct the application not to send information about the hardware to Unity (See the Unity Hardware Statistics page for more details).
Scripting Define Symbols	use this to set custom compilation flags (see the platform dependent compilation page for more details).
Allow 'unsafe' Code	Enables support for compiling 'unsafe' C# code in a pre-defined assembly (for example, *Assembly-CSharp.dll*). For Assembly Definition Files (*.asmdef*), click on one of your *.asmdef* files and enable the option in the **Inspector** window that appears.
Optimization	
.Net 2.0	.Net 2.0 libraries. Maximum .net compatibility, biggest file sizes.
.Net 2.0 Subset	Subset of full .net compatibility, smaller file sizes.
Prebake Collision__ Meshes|**Check this box to add** collision__A collision occurs when the physics engine detects that the colliders of two GameObjects make contact or overlap, when at least one has a rigidbody component and is in motion. More info See in Glossary data to meshes at build time.	
Preload Shaders	Check this box to load shaders when the player starts up.
Preloaded Assets	Set an array of **assets** to be loaded when the player starts up.

Property:	*Function:*
Vertex Compression	Vertex compression can be set per channel. You can, for instance, choose to have compression enabled for everything except positions and lightmap UVs. Whole **mesh** compression set per imported object will override the vertex compression on objects that have it set, while everything else will obey the vertex compression options/channels set here.
Optimize Mesh Data	Check this box to remove any data from meshes that is not required by the material applied to them (e.g. tangents, normals, colors, UV).

API Compatibility Level

You can choose your mono API compatibility level for all targets. Sometimes a 3rd-party .net dll will use things that are outside of the .net compatibility level that you would like to use. In order to understand what is going on in such cases, and how to best fix it, install "Reflector" for Windows.

Drag the .net assemblies for the API compatilibity level in question into Reflector. You can find these in **Frameworks/Mono/lib/mono/YOURSUBSET/**
Drag in your 3rd-party assembly.
Right-click your 3rd-party assembly and select "Analyze".
In the analysis report, inspect the "Depends on" section. Anything that the 3rd-party assembly depends on, but that is not available in the .net compatibility level of your choice, will be highlighted in red there.

XR Settings

Property:	*Function:*
XR Settings	
Virtual Reality Supported	Enable native VR support for the Unity Editor and your game builds.
XR Support Installers	
Vuforia Augmented Reality	Enable use of the Vuforia Software Development Kit. You must have a Vuforia Software License and agree to the terms of that license before the property becomes enabled.

Details

Customizing your Resolution Dialog

The Resolution Dialog, presented to end-users

You have the option of adding a custom banner image to the Screen Resolution Dialog in the Standalone Player. The maximum image size is 432 x 163 pixels. The image does not scale up to fit the screen selector, it is automatically centered and cropped.

Publishing to the Mac App Store

The property **Use Player Log** enables writing a log file with debugging information. This is useful for investigating problems with your game. However you need to disable this when publishing games for Apple's Mac App Store, as Apple may reject your submission if it is enabled. See the Unity Manual Log Files page for further information about log files.

The property **Use Mac App Store Validation** enables receipt validation for the Mac App Store. If this is enabled, your game will only run when it contains a valid receipt from the Mac App Store. Use this when submitting games to Apple for publishing on the App Store. This prevents people from running the game on a different computer to the one it was purchased on. Note that this feature does not implement any strong copy protection. In particular, any potential crack against one Unity game would work against any other Unity content. For this reason, it is recommended that you implement your own receipt validation code on top of this, using Unity's **plugin** feature. However, since Apple requires **plugin** validation to initially happen before showing the screen setup dialog, you should still enable this check, or Apple might reject your submission.

2018–08–16 Page amended with limited editorial review

2017–09–04 > Added MacOS Retina Support checkbox Added in 2017.2

Allow 'unsafe' Code checkbox added in Unity 2018.1

.NET 4.x runtime added in 2018.1

.Net 3.5 scripting runtime deprecated in Unity 2018.3

Multi-display

Leave feedback

Multi-display allows you to display up to 8 different camera views of your application on up to 8 different monitors at the same time. You could use this for PC games, arcade game machines and simple installations for public display.

Multi-display only runs in standalone mode, and is supported on Windows, Mac OS X and Linux.

Previewing Multi-display in your project

To see the different monitor displays:

Set each **Camera** to display to a specific monitor, using its **Inspector**. You can assign between 1 and 8 display monitors via the **Target Display** option (see *Fig. 1*).

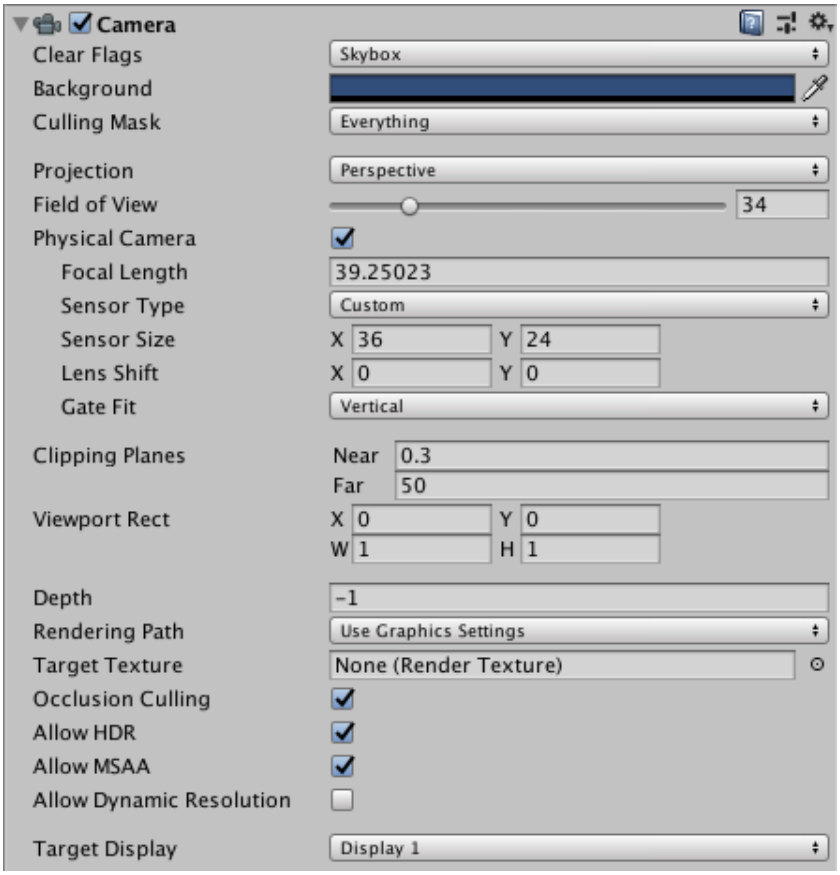


Fig. 1: Camera Inspector with Target Display option

You can then preview each display in the Game View, using the drop-down **Display** menu in the top left-hand corner of the view (see *Fig. 2*).

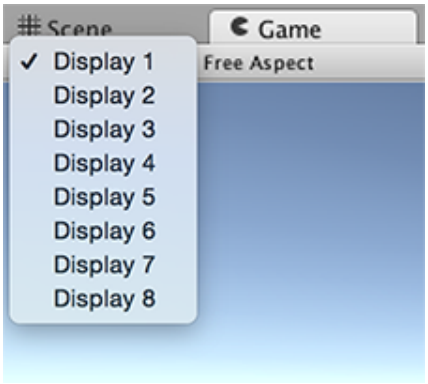


Fig 2: Display preview in the top left corner of the Game View

Activating Multi-display

The default display is one monitor, so when you run your application, you need to explicitly activate any additional displays via scripting, using **Display.Activate**. You need to explicitly activate each additional display and, once activated, you cannot deactivate them.

The best time to activate additional displays is upon creating a new **Scene**. A good way to do this is to attach a script **component** to the default Camera. Make sure you call **Display.Activate** only once during the startup. You may find it helpful to create a small initial scene to test it.

Example script

```
using UnityEngine;
using System.Collections;

public class DisplayScript : MonoBehaviour
{
    // Use this for initialization
    void Start()
    {
        Debug.Log("displays connected: " + Display.displays.Length);
        // Display.displays[0] is the primary, default display and is always ON.
        // Check if additional displays are available and activate each.
        if (Display.displays.Length > 1)
            Display.displays[1].Activate();
        if (Display.displays.Length > 2)
            Display.displays[2].Activate();
        ...
    }
    // Update is called once per frame
    void Update()
    {

    }
}
```

API support

The following **UnityEngine.Display** API functions are supported:

```
    public void Activate()
```

This activates a specific display on the current monitor's width and height. This call must be made once upon starting a new Scene. It can be called from a user script attached to a **Camera** or dummy **GameObject** in a new scene.

```
public void Activate(int width, int height, int refreshRate)
```

Windows only: This activates a specific display on a custom monitor's width and height.

Controlling monitor display positions

By default, your user's computer sorts the relative positions of its display monitors based on its X, Y virtual desktop. To override this so that your application displays without any sorting, start your application from the command line and use the command line flag:

```
-multidisplay
```

macOS

Leave feedback

This section of the User Manual contains documentation on developing for the macOS platform. Use the navigation column on the left-hand side of the page to view topics in this section.

2018–03–13 Page amended with limited editorial review

macOS Player: IL2CPP Scripting Backend

Leave feedback

IL2CPP is a fully-supported **scripting backend** that you can use as an alternative to Mono when building Projects for macOS Player.

When building a Project using IL2CPP, Unity converts IL code from **scripts** and assemblies to C++ before creating a native binary. See IL2CPP for more information.

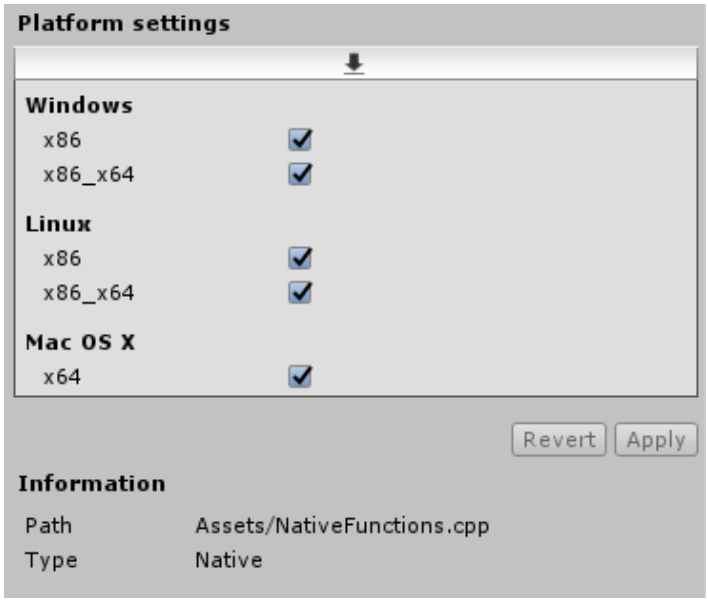
• 2018–03–13 Page published with editorial review

New feature in 2018.1

macOS Player: C++ source code plugins for IL2CPP

Leave feedback

You can add C++ (.cpp) code files directly into a Unity Project when using the IL2CPP **scripting backend**. These C++ files will act as plugins within the **Plugin Inspector**. If you configure the C++ files to be compatible with macOS Player, Unity compiles them together with C++ code that gets generated from managed assemblies. Select the appropriate **Mac OS** option in the **Platform settings** section of the **Inspector** window:



Plugin importer settings for C++ files

Because the functions are linked together with generated C++ code, there is no separate DLL to `_P/Invoke` into. Due to this, you can use the "`__Internal`" keyword in place of the DLL name, which makes it the C++ linker's responsibility to resolve the functions, rather than loading them at run time, as the following example shows:

```
[DllImport("__Internal")]
private static extern int
CountLettersInString([MarshalAs(UnmanagedType.LPWSTR)]string str);
```

You can define this kind of function in NativeFunctions.cpp as follows:

```
extern "C" __declspec(dllexport) int __stdcall CountLettersInString(wchar_t* str)
{
    int length = 0;
    while (*str++ != nullptr)
        length++;
    return length;
}
```

Because the linker resolves the function call, any error made in the function declaration on the managed side (i.e. C# code that executes under managed run time) produces a linker error rather than a run-time error. This also means that no dynamic loading needs to take place during run time, and the function is called directly from C#. This significantly decreases the performance overhead of a `P/Invoke` call.

See Also

Plugin Inspector

• 2018–03–13 Page published with editorial review

New feature in 2018.1

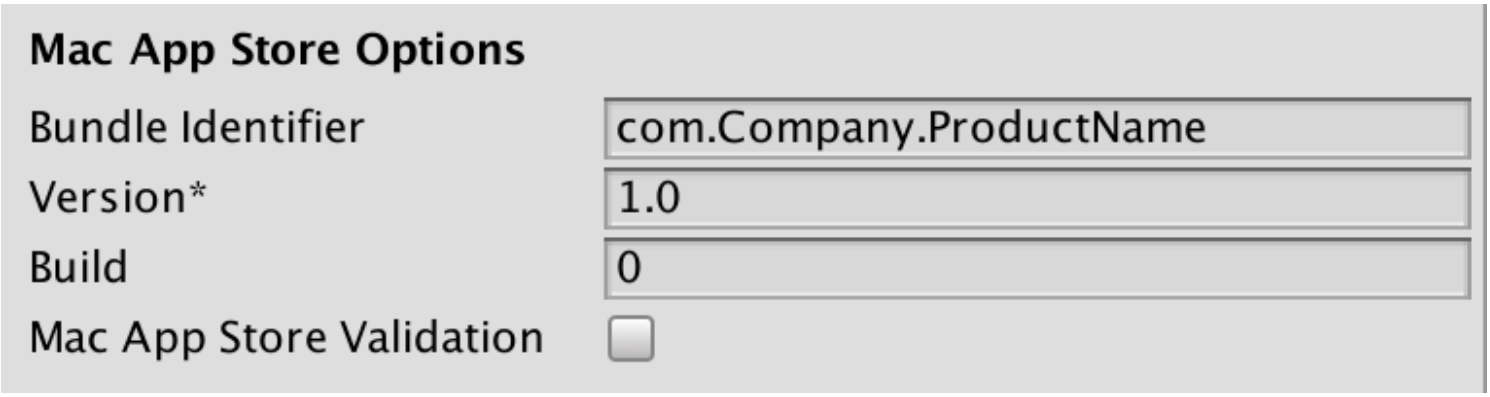
Delivering your application to the Mac App Store

Leave feedback

This page describes the process of delivering your application to the Mac App Store.

First, you need to make sure you have the correct provisioning profiles installed in your keychain: the “3rd Party Mac Developer Application” and “3rd Party Mac Developer Installer” profiles. See Apple’s developer documentation on Maintaining Your Signing Identities and Certificates to learn how to do this.

Go to **Edit** > **Project Settings** > **Player**. Open the **Other Settings** tab and navigate to **Mac App Store Options**. Unity automatically applies these settings to your app’s *info.plist* file as CF keys (see Apple’s developer documentation on Core Foundation Keys to learn more).



Mac App Store Options

Property	Function
Bundle Identifier	Enter the Bundle Identifier of your iTunesConnect App. This appears as `CFBundleIdentifier` in the associated *info.plist* file. See Apple developer documentation on CFBundleIdentifier to learn more.

Property	Function
Version | *Enter your application bundle version (for example 1.0, 2.0). This appears as* `CFBundleShortVersionString` *in the associated info.plist file.*	
See Apple developer documentation on CFBundleShortVersionString to learn more. | | *Build__The process of compiling your project into a format that is ready to run on a specific platform or platforms. More info See in Glossary* | *Enter the build number for this version of your app. This appears as* `CFBundleVersion` *in the associated* **info.plist* file.**	Enable this to ensure that your app only runs when it contains a valid receipt from the Mac App Store. This prevents people from running the game on a different device to the one it was purchased on. Only disable this setting if you have implemented your own receipt validation.
See Apple developer documentation on CFBundleVersion to learn more. | |	

Mac App Store Validation__

Tick the **Mac App Store Validation** checkbox, then build your app (**File** > **Build Settings...** > **Build**).

Next, you need to create a `GAMENAME.entitlements` file and save it in any location. The easiest way to do this is to create an empty Mac app. To do this, open Xcode, create a new project with a macOS template, go to the Capabilities bar and enable App Sandbox. This automatically generates a basic .entitlements file.

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs

<plist version="1.0">

<dict>

<key>com.apple.security.app-sandbox</key> <true/>

</dict>

</plist>
```

Note : Unity does not automatically update the *info.plist* file in versions 5.5 and earlier. Instead, you need to do this manually. See the Manual update section, below.

Open the *info.plist* file in Xcode (or any text editor), and add the following keys:

```
<key>LSApplicationCategoryType</key>

<string>{VALID APP CATEGORY, e.g.: public.app-category.kids-games }</string>

<key>CFBundleSignature</key>

<string>{4 LETTER CREATOR CODE, e. g.:  GMAD }</string>
```

If you're using Xcode 8.0+, you also need to add these fields to the *info.plist* file:

```
<key>CFBundleSupportedPlatforms</key>

    <array>

        <string>MacOSX</string>

    </array>
```

Next, fix read permissions on all the content in the .app. To do this, type the following into the macOS Terminal:

```
chmod -R a+xr "/path/to/GAMENAME.app"
```

Sign the .App with the .entitlements you created earlier. To do this, type the following into the macOS Terminal:

```
codesign -f --deep -s '3rd Party Mac Developer Application: DEVELOPER NAME' --e
```

Build the installer/pkg. To do this, type the following into the macOS Terminal:

```
productbuild --component GAMENAME.app /Applications --sign "3rd Party Mac Devel
```

Finally, use the Xcode ApplicationLoader to submit your app.

Optional additions

Manual info.plist update

If you're using a Unity version older than 5.6, you need to add the bundle identifier and version string manually, after building your app.

Modify the following sections of the `info.plist` file to make it suitable for your app:

```
<key>CFBundleDevelopmentRegion</key>

<string>{YOUR REGION}</string>

<key>CFBundleGetInfoString</key>

<string>{DESCRIPTIVE INFO}</string>

<key>CFBundleIdentifier</key>

<string>com.{YOUR COMANY}.{YOUR APP NAME}</string>

<key>CFBundleName</key>

<string>{YOUR APP NAME}</string>

<key>CFBundleShortVersionString</key>

<string>{VERSION NUMBER, e.g. 1.0.0}</string>

<key>CFBundleSignature</key>

<string>{4 LETTER CREATOR CODE, e. g.:  GMAD }</string>

<key>CFBundleVersion</key>

<string>{VERSION NUMBER, e.g. 100}</string>
```

Custom Icons

By default, Unity downsizes the icon image that you specified in **Player Settings** > **Default Icons** to generate an .icns file. This defines how your app's icon appears in the macOS Finder and in the OS dock. However, you can

replace it with a custom icon set if you want to.

Make a folder and name it *UnityPlayer.iconset* (or whichever name is set in your *info.plist*'s `CFBundleIconFile/Icon File` field) and place the following image names inside. Note that this folder must have the .iconset extension.

```
icon_16x16.png

icon_16x16@2x.png

icon_32x32.png

icon_32x32@2x.png

icon_128x128.png

icon_128x128@2x.png

icon_256x256.png

icon_256x256@2x.png

icon_512x512.png

icon_512x512@2x.png
```

Make sure that the `@2x.png` images are double the size stated in the file name. For example, the image `512x512@2x.png` contains an image that is 1024x1024. From the macOS Terminal, navigate to the directory where the .iconset directory is located, and enter the following command:

```
iconutil -c icns UnityPlayer.iconset
```

Finally, right-click the .app file and, select **Show Contents**, and replace the *iconset.icns* with the one you created earlier.

2017–05–18 Page published with editorial review

Updated features in 5.6

Apple TV

Leave feedback

This section of the User Manual contains documentation on developing for the Apple TV platform. Use the navigation column on the left-hand side of the page to view topics in this section.

Building games for Apple TV

Leave feedback

This manual page is primarily aimed at helping developers transition from **iOS** to tvOS. The Apple TV platform (also known as tvOS) builds on the foundation of the iOS platform while creating new paradigms and challenges for game developers. Deploying an existing mobile game on tvOS is just one click away, but game content often needs to be adapted to work correctly with Unity's new input controls and the fact that the game is displayed on a big screen.

Prerequisites

To develop for tvOS you need the following:

A 4th generation Apple TV device (you also need a USB C <-> USB 3.0 cable, which is not included with the consumer package).
Xcode 7.1 or later.
You need to set up provisioning for this device in the same way as for iOS devices. It is recommended to create an empty tvOS app with Xcode to test that provisioning works correctly.

Points to note before starting

Many iOS plugins aren't compatible with Apple TV, because it only supports a subset of the iOS framework. It's recommended that you create a separate branch or copy of your game and do porting to Apple TV there. Reach out to **plugin** providers to ask them to update incompatible plugins.
If your game uses more than 200 MB on disk, you should break it into smaller parts and use **On Demand Resources**. For On Demand Resources support in Unity please refer to the *On Demand Resources* section below. Note that Bitcode is included with tvOS builds, which adds ~130 MB to your executables. This added size is not accounted for in distribution size, as it is be stripped by App Store servers. You can estimate Bitcode size by analyzing LLVM sections in your executable from the command line with `otool -l`.

Implementing input

The Apple TV Remote (Siri Remote) serves as a multi-purpose input device, working both as a traditional menu navigation controller, **game controller**, gyro and acceleration sensor, and as a touch gesture device. Apple TV Remote input is minimally processed by Unity and mostly routed to corresponding Unity APIs.

Usually, each game needs slight adjustment of its input scheme to leverage unique Apple TV Remote input features. Some games would benefit from treating it as a traditional game controller, with one analog axis and an extra action button, while others would benefit from using the accelerometer (for example, for steering purposes). It is recommended that you experiment with various schemes when porting a game to tvOS.

Here are some technical details on accessing specific TV Remote features:

If you have not yet added Made For iOS (MFi) Game Controller support to your game, see the dedicated MFi Game Controller page. Use the mappings listed there while setting up your custom action mappings in the Unity Editor (Edit > Project Settings > Input**).
The Apple TV Remote touch area is mapped both to `Input.touches` (`Touch.type` is set to `Indirect` and is ignored by the Unity GUI), and the usual Joystick Input API (e.g. `Input.GetAxis("Horizontal");`)
The Apple TV Remote acceleration and gyroscope are mapped accordingly to `Input.acceleration` and `Input.gyro`. `Input.acceleration` internally is derived from gyroscope API and might have some instabilities. Unfortunately there is no dedicated accelerometer API in the tvOS SDK. `Input.gyro.attitude` is derived from the gravity vector, and thus lacks rotation around the axis parallel to the gravity vector. The same applies for `Input.gyro.rotationRate`.
The Pause/Play button on the remote is mapped to button "X" (which is then mapped to joystick button 15).
The Apple TV Remote touch area click is mapped to button "A" (which is then mapped to joystick button 14).

The Menu button has special behavior on this device; a long press invokes the tvOS task switcher. This behavior cannot be overridden. Short taps can be processed two ways:
a) Returning to the tvOS system home screen (if `UnityEngine.Apple.TV.Remote.allowExitToHome` is **true**)
b) Letting your app respond to taps (mapped to button "Pause" / joystick button 0), when `UnityEngine.Apple.TV.Remote.allowExitToHome` is **false**. This the default behavior.
Your app should switch between **a)** and **b)** based on the current state of your game. If the user is interacting with the top menu, enable behavior **a)**; if they are interacting in real time with the game you should enable behaviour **b)** and invoke in the in-game pause menu when this button is pressed.
The Apple TV Remote also generates directional pad up/down/left/right button presses when you swipe to the edge of the remote. Refer to the manual page for iOS Game Controllers for the mappings.
The Apple TV Remote operational modes can be controlled via a dedicated API:
`UnityEngine.Apple.TV.Remote.allowExitToHome`
`UnityEngine.Apple.TV.Remote.allowRemoteRotation`
`UnityEngine.Apple.TV.Remote.reportAbsoluteDpadValues`
`UnityEngine.Apple.TV.Remote.touchesEnabled`
Two further wireless Made For iOS (MFi) game controllers may be connected to an Apple TV device, effectively turning it into **game console**. Your game may use them in the same way as iOS MFi controllers (as mentioned above), though your game should be still playbable with the Apple TV Remote alone. Currently, the number of additional controllers is limited to two; this is a documented tvOS system limitation.
Warning: due to the Apple TV Remote "Menu" button being reported as joystick button 0 when `UnityEngine.Apple.TV.Remote.allowExitToHome` is set to **false**, and the default Input Manager binding "Submit" virtual button being mapped to the same joystick button 0, this button triggers actions on **UI** elements when pressing the "Menu" button. To work around this issue, remove or modify the "Submit" virtual button bindings in the Input Manager.

Setting up Unity GUI navigation via Apple TV Remote

Open the Input Manager in the Unity Editor. Find the first occurrence of the "Submit" virtual input, expand it and change its "Alt Positive Button" to "joystick button 14".
Select the EventSystem game object in your **scene**. In the **Inspector**, find the EventSystem **component** and populate the "First Selected" field with the UI game object that should receive initial focus. You may need to check the "Force input module" flag in the "Standalone Input Module" **component**.
This allows you to navigate your UI via the keyboard while running in the Editor and via Apple TV Remote swipes and full stop click when running on your device.

Note: Apple TV Remote navigation does not work while running in the TV Simulator.

Adding leaderboard resources to Xcode project

Game Center requires custom visual resources to be provided for its native leaderboard UI. Here are quick instructions on how to set them up in Xcode:

Select Images.xcassets in Xcode project
Right-click under listed files, and from the menu pick **Game Center > New AppleTV Leaderboard**.
Add your images.
Select Leaderboard and on the right pane pick Edit View. Find the "Identifier" field and enter your leaderboard ID there.
If after these modifications your **asset** compilation starts to fail, try disabling "On Demand Resources" in Xcode "Build Settings".

Implementing On Demand Resources support

tvOS has strict requirements on how much disk space your application can reserve. The main application installation bundle size can not exceed 200 MB. However, the limits for additional downloadable content are much higher (up to 2GB for in-use assets and up to 20GB of total downloadable content). Apple recommends On Demand Resources (ODR) for tvOS downloadable content, as it enables the best disk space management strategies for tvOS. Unity supports **ODR** via Asset Bundles. An **ODR** implementation guide can be found in our dedicated blogpost on the subject.

Known limitations

On-screen keyboard is limited to single line entries.
tvOS Simulator does not emulate the Apple TV Remote as a Game Controller, thus making its input inaccessible to games.

WebGL

Leave feedback

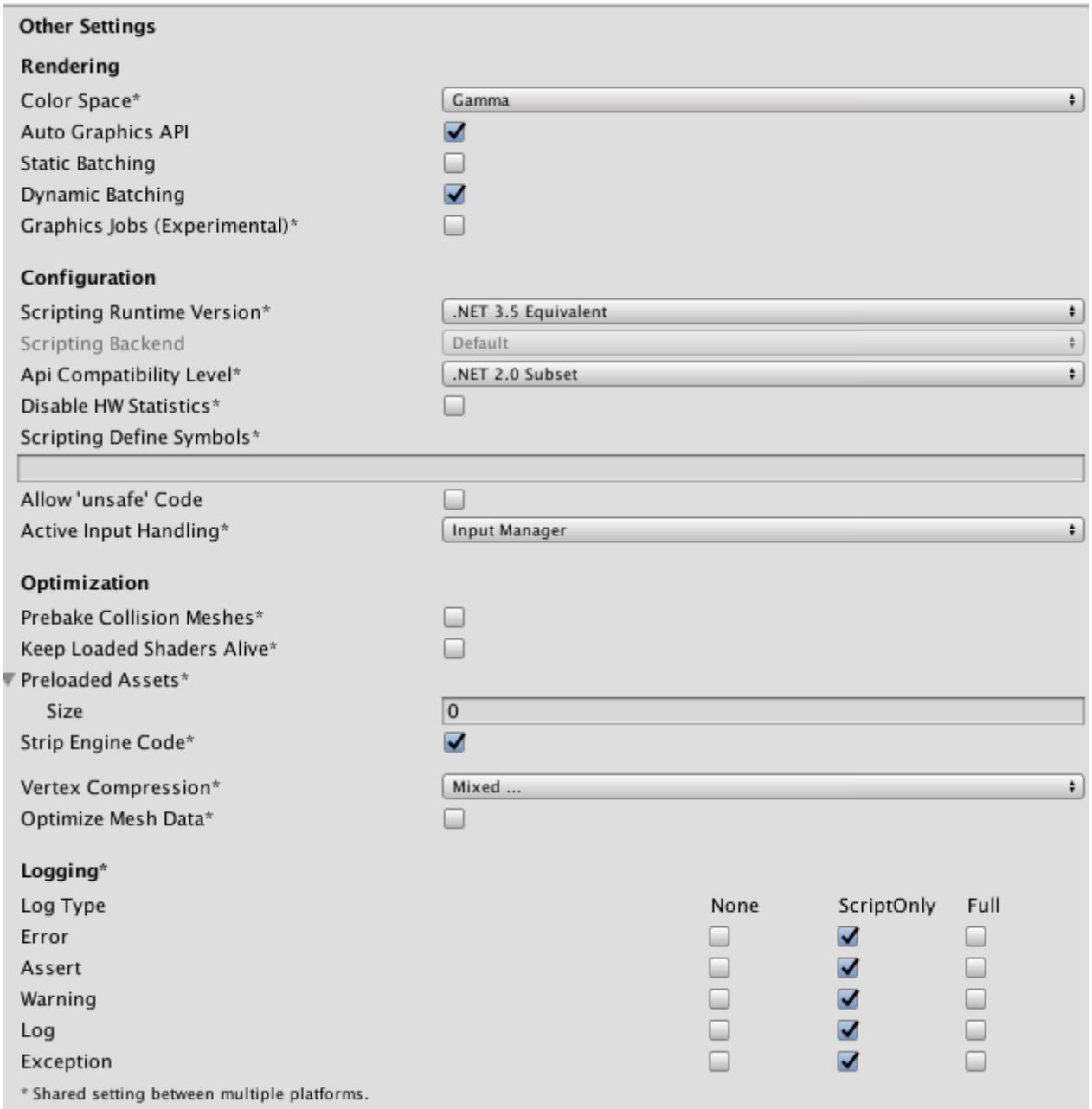
This section of the User Manual contains documentation on developing for the **WebGL** platform. Use the navigation column on the left-hand side of the page to view topics in this section.

WebGL Player Settings

Leave feedback

This page details the **Player Settings** specific to **WebGL**. A description of the general Player Settings can be found here.

Other settings



Property:	*Function:*
Rendering	
Rendering Path	The rendering path enabled for the game.
Auto Graphics API	Uncheck this if you want to manually select which graphics API is used. When checked, Unity includes WebGL2.0, with WebGL1.0 as a fallback for browsers that don't support it. When unchecked, you can manually pick and reorder the graphics APIs.

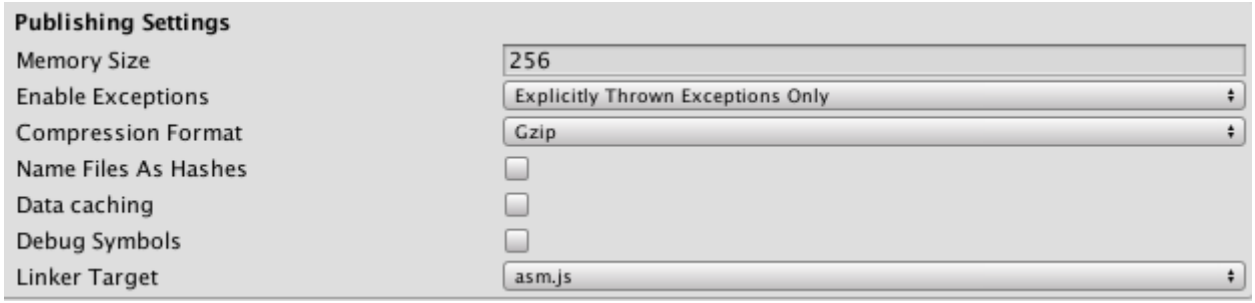
Property:	*Function:*
Static Batching	Check to enable static batching.
Dynamic Batching	Check to enable dynamic batching.
Configuration	
Scripting Runtime Version	Choose which .NET runtime to use in your project. For more details, see Microsoft's .NET documentation.
.NET 3.5 Equivalent	A .NET runtime which implements the .NET 3.5 API. This functionality is deprecated, and should no longer be used. Please use .NET 4.
.NET 4.x Equivalent	A .NET runtime which implements the .NET 4 API. This API is newer than .NET 3.5, and as such, it offers access to more APIs, is compatible with more external libraries, and supports C# 6. This is the default scripting runtime.
Scripting Backend	Scripting backend is grayed out because there is only one scripting backend on WebGL.
Disable HW Statistics	When this is unchecked, the application sends information about the hardware to Unity (See hwstats page for more details).
Scripting Define Symbols	Custom compilation flags (see the platform dependent compilation page for details).
Allow 'unsafe' Code	Enables support for compiling 'unsafe' C# code in a pre-defined assembly (for example, *Assembly-CSharp.dll*). For Assembly Definition Files (*.asmdef*), click on one of your *.asmdef* files and enable the option in the **Inspector** window that appears.
Optimization	
Api Compatibility Level	Specifies active .NET API profile. See below.
.Net 2.0	.Net 2.0 libraries. Maximum .net compatibility, biggest file sizes
.Net 2.0 Subset	Subset of full .net compatibility, smaller file sizes
Prebake	
Collision__ Meshes__	Enable **collision mesh** baking during the build.
Preload Shaders	Enable **shader** preloading.
Preload Assets	Enable **asset** preloading. Specify the size of assets to preload.
Strip Engine Code	Enable code stripping for WebGL.
Vertex Compression	
Optimize Mesh Data	Remove any data from meshes that is not required by the material applied to them (tangents, normals, colors, UV).

API compatibility level

You can choose your mono API compatibility level for all targets. Sometimes a 3rd party .net dll will use things that are outside of the .net compatibility level that you would like to use. In order to understand what is going on in such cases, and how to best fix it, get "Reflector" on windows.

Drag the .net assemblies for the API compatilibity level in question into reflector. You can find these in Frameworks/Mono/lib/mono/YOURSUBSET/
Also drag in your third-party assembly.
Right click your third-party assembly, and select **Analyze**.
In the analysis report, inspect the **Depends on** section. Anything that the third-party assembly depends on, but is not available in the .net compatibility level of your choice, will be highlighted in red there.

Publishing settings



Property:	*Function:*
Memory Size	Sets the memory available to the WebGL runtime, given in megabytes. You should choose this value carefully: if it is too low, you will get out-of-memory errors because your loaded content and **scenes** won't fit into the available memory. However, if you request too much memory then some browser/platform combinations might not be able to provide it and consequently fail to load the player. See here for details.
Enable Exceptions	Enables exceptions support which allows you to specify how unexpected code behavior (generally considered errors) is handled at run time. There are four options: None, Explicitly Thrown Exceptions Only, Full Without Stacktrace and Full With Stacktrace. See the Building and running a WebGL project page for details.
Compression Format	Release build files **compression** format: gzip, brotli or none. Note that this option does not affect **Development builds**.
Name Files As Hashes	Use MD5 hash of the uncompressed file contents as a filename for each file in the build.
Data caching	Enable this to automatically cache your contents asset data on the users machine so it will not have to be re-downloaded on subsequent runs (unless the contents have changed). Caching is implemented using the IndexedDB API provided by the browser. Some browsers may implement restrictions around this, such as asking the user for permission to cache data over a specific size.
Debug Symbols	Preserve debug symbols and perform demangling of the stack trace when an error occurs. For release builds all the debug information is stored in a separate file which is downloaded from the server on demand when an error occurs. Development builds always have demangling support embedded in the main module and therefore are not affected by this option.

Property:	*Function:*
Linker Target	Specify which build type to generate. **asm.js** is widely supported across browsers, while **WebAssembly** is a new and efficient format for the web. Selecting **Both** generates WebAssembly and asm.js formats. Then at run time, WebAssembly is used if it's supported by the browser, otherwise it will fall back to asm.js.

Further information about WebGL Publishing Settings can be found in the WebGL Building and Running page.

.Net 3.5 scripting runtime deprecated in Unity 2018.3

2018–08–16 Page amended with no editorial review

Publishing settings updated in Unity 2017.3

Allow 'unsafe' code checkbox added in Unity 2018.1

.NET 4.x runtime added in 2018.1

Getting started with WebGL development

Leave feedback

What is Unity WebGL?

The **WebGL** build option allows Unity to publish content as JavaScript programs which use HTML5 technologies and the **WebGL rendering** API to run Unity content in a web browser. To build and test your content for **WebGL**, choose the **WebGL** build target in the **Build Player** window, and click **Build & Run**.

Technical overview

To run in WebGL, all code needs to be JavaScript. We use the emscripten compiler toolchain to cross-compile the Unity runtime code (written in C and C++) into asm.js JavaScript. asm.js is a very optimizable subset of JavaScript which allows JavaScript engines to AOT-compile asm.js code into very efficient native code.

To convert the .NET game code (your C# scripts) into JavaScript, we use a technology called IL2CPP. IL2CPP takes .NET bytecode and converts it to corresponding C++ source files, which is then compiled using emscripten to convert your **scripts** to JavaScript.

Platform support

Unity WebGL content is supported in the current versions of most major browsers on the desktop, however there are differences in the level of support offered by the different browsers. Mobile devices are not supported by Unity WebGL.

Not all features of Unity are available in WebGL builds, mostly due to constraints of the platform. Specifically:

Threads are not supported due to the lack of threading supporting in JavaScript. This applies to both Unity's internal use of threads to speed up performance, and to the use of threads in script code and managed dlls. Essentially, anything in the `System.Threading` namespace is not supported.

WebGL builds cannot be debugged in Visual Studio. See: Debugging and trouble shooting WebGL builds.

Browsers do not allow direct access to IP sockets for **networking**, due to security concerns. See: WebGL Networking.

The WebGL graphics API is equivalent to OpenGL ES 2.0 and 3.0, which has some limitations. See: WebGL Graphics.

WebGL builds use a custom backend for Audio, based on the **Web Audio** API. This supports only basic audio functionality. See: Using Audio in WebGL.

WebGL is an AOT platform, so it does not allow dynamic generation of code using `System.Reflection.Emit`. This is the same on all other IL2CPP platforms, **iOS**, and most consoles.

2018–03–19 Page amended with limited editorial review

MonoDevelop replaced by Visual Studio from 2018.1

WebGL Browser Compatibility

Leave feedback

Unity **WebGL** supports all major desktop browsers to some degree. However, the level of support and the expected performance varies between different browsers. See the table below for an overview of browser features of interest to Unity **WebGL** content, and which browsers support them.

Note that Unity WebGL content is not currently supported on mobile devices. It may still work, especially on high-end devices, but many current devices are not powerful enough and don't have enough memory to support Unity WebGL content well. For this reason, Unity WebGL shows a warning message when trying to load content on mobile browsers (which can be disabled if needed).

Note that this compatibility table is valid for the specific versions of the browsers as stated. Support should continue for future versions, but may not be stable in previous versions.

Desktop browser compatibility table

	Mozilla Firefox 52	Google Chrome 57	Apple Safari 11	MS Edge 16
WebGL Support	**Yes** GPU blacklists apply. WebGL may be unsupported for specific older graphics cards. Details available on the Mozilla wiki page on Blocklisting/Blocked Graphics Drivers and the Khronos wiki page on Blacklists and Whitelists.	**Yes** GPU blacklists apply. WebGL may be unsupported for specific older graphics cards. Details available on the Mozilla wiki page on Blocklisting/Blocked Graphics Drivers and the Khronos wiki page on Blacklists and Whitelists.	**Yes** Safari 8 and higher	**Yes**
Web Audio (See Web Audio) The Web Audio API is required to play sound in Unity WebGL content.	**Yes**	**Yes**	**Yes**	**Yes**
Full-screen support (See Full-screen support)	**Yes**	**Yes**	**Yes** Safari 10.1 or newer	**Yes**
Cursor locking support (see Cursor Locking support)	**Yes**	**Yes**	**Yes**	**Yes** Edge 13 and newer.
Gamepad support (See Gamepad support)	**Yes**	**Yes**	**Yes**	**Yes**

Desktop browser compatibility table

IndexedDB Required for local storage as used by the Data Caching feature, the PlayerPrefs class, and WWW.LoadFromCacheOrDownload.	**Yes** Firefox up to version 42 does not support IndexedDB for content running in an iFrame. Firefox 43 and higher fixes this.	**Yes**	**Yes** Safari does not support IndexedDB for content running in an iFrame.	**Yes**
WebSockets Required for Networking.	**Yes**	**Yes**	**Yes**	**Yes**
WebRTC Required by the WebCamTexture class.	**Yes**	**Yes**	**No**	**Yes**
WebGL 2.0 (See WebGL 2.0)	**Yes** Firefox 51 and newer	**Yes** Chrome 56 and newer	**No**	**No**
asm.js AOT compilation asm.js is a susbset of JavaScript for which a browser can specifically optimize. Browsers which implement asm.js support may be able to run Unity WebGL content faster, because Unity uses asm.js.	**Yes**	**No**	**No**	**Yes**
WebAssembly WebAssembly (or wasm) is a new portable, size-efficient and load-time-efficient format suitable for compilation to the web.	**Yes** Firefox 52 and newer.	**Yes** Chrome 57 and newer.	**Yes** Safari 11 or newer	**Yes** Edge 16 or newer
Large-Allocation Http header Helps browsers make sure enough memory is available to load your content (See Large-Allocation Http Header)	**Yes** Firefox 53 and newer.	**No**	**No**	**No**
Brotli Compression Reduces the build size (see Brotli compression)	**Yes**	**Yes**	**Yes** Safari 11 or newer	**Yes**

2018–09–14 Page amended with limited editorial review

Brotli compression first documented on this page in User Manual 5.6

Building and running a WebGL project

Leave feedback

When you build a WebGL project, Unity creates a folder with the following files:

An *index.html* file which browsers can navigate to load your content.

A *TemplateData* folder (when building with the default template) containing the build logo, loading bar and other template **Assets**. The build template folder is normally used to customize the appearance of the build while loading. See the User Manual page on WebGL templates for more information.

A *Build folder* containing your generated build output files.

The *Build* folder contains the following files (the *MyProject* file name represents the name of your project).

A *UnityLoader.js* JavaScript file containing the code needed to load up the Unity content in the web page.

A *MyProject.json* JSON file containing all the necessary information about your build. The URL of this JSON file is provided as an argument for the Unity Loader when the build is instantiated. JSON file contains URLs of all the other build files, which can be absolute or relative to the location of the JSON file. JSON may contain additional Module parameters, such as the splash screen style or the initial size of the memory heap.

A *MyProject.asm.framework.unityweb* file containing the asm.js runtime and JavaScript **plugins**.

A *MyProject.asm.code.unityweb* file containing the asm.js module for your player.

A *MyProject.asm.memory.unityweb* file containing a binary image to initialize the heap memory for your player.

A *MyProject.data.unityweb* file containing the Asset data and **Scenes**.

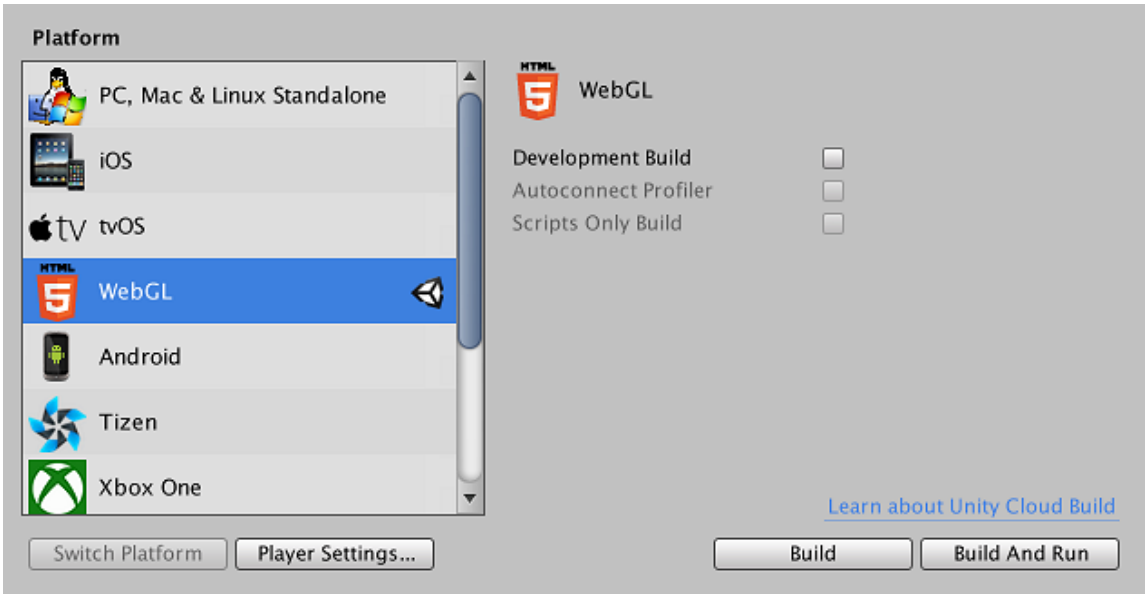
The contents of the **.unityweb* files in the *Build* folder may be compressed with gzip, brotli or may be uncompressed, depending on the Publishing Settings. See documentation on deploying compressed builds for more information

You can view your WebGL player directly in most browsers by opening the *index.html* file. However, for security reasons, Chrome places restrictions on scripts opened from local file URLs, so this technique does not work in Chrome. To work round Chrome's restrictions, use Unity's **Build & Run** command (**File** > **Build & Run**); the file is then temporarily hosted in a local web server and opened from a local host URL. Alternatively, you can run Chrome with the `--allow-file-access-from-files` command line option which allows it to load content from local file URLs. This is required on a PC to allow execution of the build.

On some servers you need to explicitly make *.unityweb* files accessible, because the server needs to provide these files to clients.

Build player options

Access the WebGL options via the in the **Build Settings** dialog box. (menu: **File** > **Build Settings...**). In the dialog box, select **WebGL** from the **Platform** list, then select **Player Settings...**.



Development Build

When you check the **Development Build** checkbox, Unity generates a development build which has Profiler support and a development console to see errors. In addition, development builds do not compress content (that is, content is not minified); they maintain JavaScript in human-readable form, preserving function names so you get useful stack traces for errors. Note, however, that this means development builds are very large, and too large to distribute.

Use pre-built Engine

This option only appears when you have the **Development Build** checkbox checked. Use the **Use pre-built Engine** option in the Build Settings dialog box to speed up build iteration time during development. When this option is enabled, Unity rebuilds only the managed code and then links dynamically with the pre-build Unity engine, so the project is rebuilt approximately 30–40% faster. Note that this type of build is only suited for development purposes, as it always produces unstripped engine code. Due to the dynamic linking overhead, the performance of this type of build is slower than a normal build.

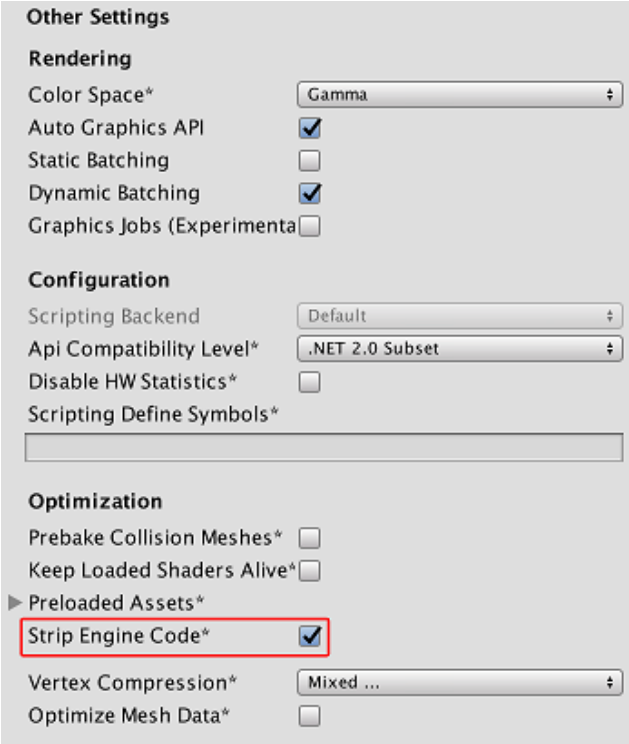
Autoconnect Profiler

This option can only be enabled when you have the **Development Build** checkbox checked. Check the **Autoconnect Profiler** option to profile your Unity WebGL content. For WebGL, it is not possible to connect the Profiler to a running build as on other platforms, so you have to use this option to connect the content to the Editor. This is because the Profiler connection is handled using WebSockets on WebGL, but a web browser only allows outgoing connections from the content.

Player Settings

WebGL has some additional options in the **PlayerSettings** Inspector window (menu: **Edit** > **Project Settings** > **Player**).

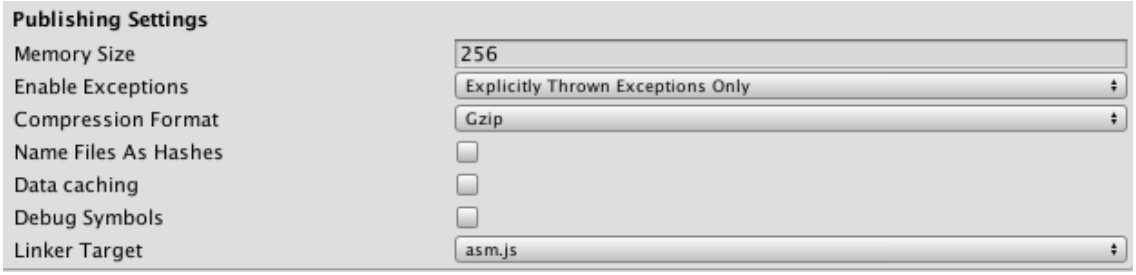
Other Settings



Strip Engine Code

Open **Other Settings** to access the **Strip Engine Code** option. This option is checked by default to enable code stripping for WebGL. With this option checked, Unity does not include code for any classes you don't use. For example, if you don't use any physics **components** or functions, the whole **physics engine** is removed from your build. See the Stripping section below for more details.

Publishing Settings



Memory Size

Open **Publishing Settings** to access the **Memory Size** field. Here, you can specify how much memory (in MB) the content should allocate for its heap. If this value is too low, an "out of memory" error message appears. This means your loaded content and Scenes cannot fit into the available memory. However if this value too high, your content might fail to load in some browsers or on some machines, because the browser might not have enough available memory to allocate the requested heap size. This value is written to a property named `TOTAL_MEMORY` in the generated .json file, so if you want to experiment with this setting without having to rebuild your project, you can either edit the .json file or provide the updated `TOTAL_MEMORY` value as an additional WebGL instantiation parameter. See the User Manual page on WebGL memory usage for more details.

Enable Exceptions

Open **Publishing Settings** to access **Enable Exceptions**. **Enable Exceptions** allows you to specify how unexpected code behavior (generally considered errors) is handled at run time. It has these options:

None: Select this if you don't need any exception support. This gives the best performance and smallest builds. With this option, any exception thrown causes your content to stop with an error in that setting.

Explicitly Thrown Exceptions Only (default): Select this to capture exceptions which are explicitly specified from a `throw` statement in your **scripts** and to also ensure `finally` blocks are called. Note that selecting this option makes the generated JavaScript code from your scripts longer and slower; This might only be an issue if scripts are the main bottleneck in your project.
Full Without Stacktrace: Select this option to capture:
Exceptions which are explicitly specified from `throw` statements in your scripts (the same as in the *Explicitly Thrown Exceptions Only* option)
Null References
Out of Bounds Array accesses
Full With Stacktrace: This option is similar to the option above but it also captures Stack traces. Unity generates these exceptions by embedding checks for them in the code, so this option decreases performance and increases browser memory usage. Only use this for debugging, and always test in a 64-bit browser.
Select **Publishing Settings** to access **Data Caching**. Select this to enable automatic local caching of your player data. This option sets asset storage as a local cached in the browser's IndexedDB database, so that assets won't have to be downloaded again in subsequent runs of your content. Note that different browsers have different rules on allowing IndexedDB storage; browsers may ask the user for permission to store the data, and your build may exceed a size limit defined by the browser.

Distribution size

When publishing for WebGL, it is important to keep your build size low so users get reasonable download times before the content starts. For generic tips on reducing asset sizes, see documentation on Reducing the file size of the build.

Hints and tips specific to WebGL

Specify the **Crunch** texture **compression** format for all your compressed textures in the Texture Importer.

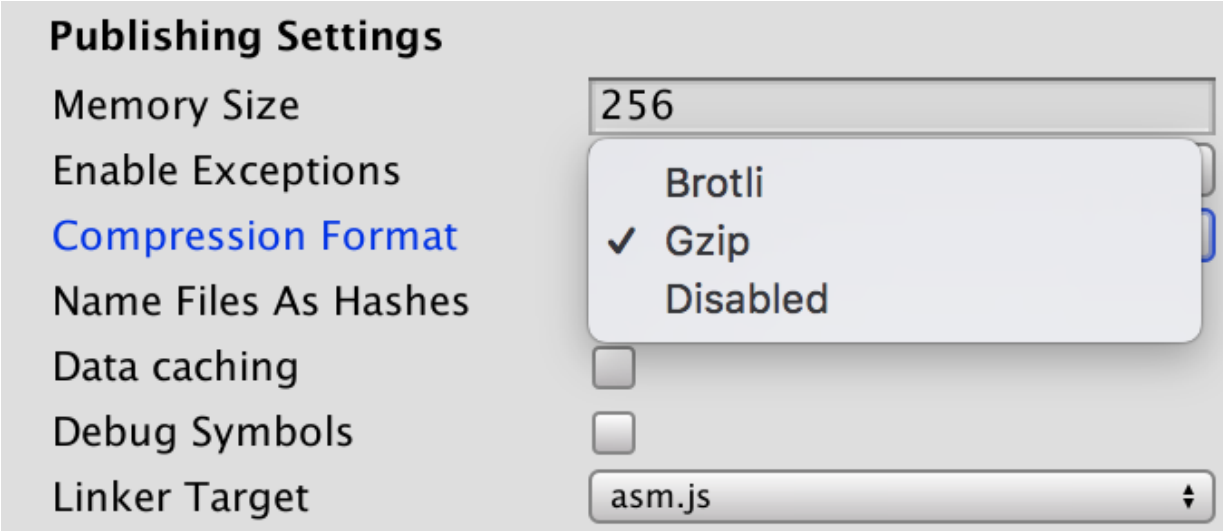
Don't deploy development builds; they are not compressed or minified, and so have much larger file sizes.

Go to **PlayerSettings** (menu: **Edit** > **Project Settings** > **Player**), open **Publishing Settings** and set **Enable Exceptions** to **None** if you don't need exceptions in your build.

Go to **PlayerSettings** (menu: **Edit** > **Project Settings** > **Player**), open **Other Settings** and enable **Strip Engine Code** to ensure an efficient build.

Take care when using third-party managed dlls, as they might include a lot of dependencies and so significantly increase the generated code size.

If you make a release build, Unity compresses the build output files according to the **Compression Format** selected in the WebGL **PlayerSettings** > **Publishing Settings** window.



See documentation on Deploying compressed builds for more info on these options, and on how to publish builds with them.

AssetBundles

Since all your Asset data needs to be pre-downloaded before your content starts, you should consider moving Assets out of your main data files and into AssetBundles. That way, you can create a small loader Scene for your content which loads quickly. It then dynamically loads Assets on-demand as the user proceeds through your content. AssetBundles also help with Asset data memory management: You can unload Asset data from memory for Assets you don't need any more by calling AssetBundle.Unload.

Some considerations apply when using AssetBundles on the WebGL platform:

When you use class types in your AssetBundle which are not used in your main build, Unity may strip the code for those classes from the build. This can cause a fail when trying to load Assets from the AssetBundle. Use BuildPlayerOptions.assetBundleManifestPath to fix that, or see the section on *Stripping* , below, for other options.

WebGL does not support threading, but http downloads only become available when they have finished downloading. Because of this, Unity WebGL builds need to decompress AssetBundle data on the main thread when the download is done, blocking the main thread. To avoid this interruption, LZMA AssetBundle compression is not available for AssetBundles on WebGL. AssetBundles are compressed using LZ4 instead, which is de-compressed very efficiently on-demand. If you need smaller compression sizes than LZ4 delivers, you can configure your web server to use gzip or Brotli compression (on top of LZ4 compression) on your AssetBundles. See documentation on Deploying compressed builds for more information on how to do this.

AssetBundle caching using WWW.LoadFromCacheOrDownload is supported in WebGL using the IndexedDB API from the browser to implement caching on the user's computer. Note that IndexedDB may have limited support on some browsers, and that browsers may request the user's authorization to store data on disk. See documentation on WebGL browser compatibility for more information.

Stripping

Unity removes all unused code from your build by default. You can change this via the **PlayerSettings** Inspector window (menu: **Edit** > **Project Settings** > **Player**): Select **Other Settings** to access the **Strip Engine Code** option. It is better to build with stripping enabled.

With code stripping, Unity scans your project for any `UnityObject`-derived classes used (either by being referenced in your script code, or in the serialized data in your Scenes). It then removes from the build any Unity subsystems which have none of their classes used. This makes your build have less code, resulting in both smaller downloads and less code to parse (so code runs faster and uses less memory).

Issues with code stripping

Code stripping might cause issues with your project if it strips code which is actually necessary. This can be the case when you load AssetBundles at run time which contain classes that are not included in the main build, and have therefore been stripped from the project. Error messages appear in your browser's JavaScript console when this happens (possibly followed by more errors). For example:

```
Could not produce class with ID XXX
```

To troubleshoot these errors, look up the ID (such as `XXX` in the example above) in the Class ID Reference to see which class it is trying to create an instance of. In such cases, you can force Unity to include the code for that class in the build, either by adding a reference to that class to your scripts or to your Scenes, or by adding a *link.xml* file to your project.

Below is an example which makes sure that the Collider class (and therefore the Physics module) gets preserved in a project. Add this XML code to a file called *link.xml*, and put that file into your *Assets* folder.

```
<linker>
    <assembly fullname="UnityEngine">
        <type fullname="UnityEngine.Collider" preserve="all"/>
    </assembly>
</linker>
```

If you suspect that stripping is causing problems with your build, you can also try disabling the **Strip Engine Code** option during testing.

Unity does not provide a convenient way to see which modules and classes are included in a build, which would allow you to optimize your project to strip well. However, to get an overview of included classes and modules, you can look at the generated file *Temp/StagingArea/Data/il2cppOutput/UnityClassRegistration.cpp* after making a build.

Note that the **Strip Engine Code** option only affects Unity engine code. IL2CPP always strips byte code from your managed dlls and scripts. This can cause problems when you need to reference managed types dynamically through reflection rather than through static references in your code. If you need to access types through reflection, you may also need to set up a *link.xml* file to preserve those types. See the documentation page on iOS Build size optimization for more information on *link.xml* files.

Moving build output files

To change the location of your *Build* folder, change the URL of the JSON file (the second argument of the *UnityLoader.instantiate*) in the index.html file.

To change the location of the files inside the Build folder, change their URLs (that is, *dataUrl*, *asmCodeUrl*, *asmMemoryUrl*, and *asmFrameworkUrl*) in the JSON file. All non-absolute URLs in the JSON file are treated as URLs relative to the location of the JSON file. You can specify URLs on external servers for these if you want to host your files on a content distribution network (CDN), but you need to make sure that the hosting server has enabled Cross Origin Resource Sharing (CORS) for this to work. See the manual page on WebGL networking for more information about CORS.

Incremental builds

The C++ code generated for your project by IL2CPP is compiled incrementally; that is, only generated C++ code that has changed since the last build is compiled again. Unchanged source code re-uses the same object files generated for the previous build. The object files used for incremental C++ builds are stored in the *Library/il2cpp_cache* directory in your Unity project.

To perform a clean, from-scratch build of the generated C++ code which doesn't use incremental compiling, delete the *Library/il2cpp_cache* directory in your Unity project. Note that if the Unity Editor version differs from the one used for the previous WebGL build, Unity does a clean, from-scratch build automatically.

2017–08–25 Page amended with limited editorial review

Full Without Stacktrace added in Unity 2017.3

WebGL: Deploying compressed builds

Leave feedback

When you build a WebGL project in release mode (see Publishing builds), Unity compresses your build output files to reduce the download size of your build. You can choose the type of compression it uses from the Compression Format options in Publishing Settings (menu: **Edit** > **Project Settings** > **Player** > **Publishing Settings**):

gzip: This is the default option. gzip files are bigger than Brotli files, but faster to build, and natively supported by all browsers over both http and https.

Brotli: Brotli **compression** offers the best **compression** ratios. Brotli compressed files are significantly smaller than gzip, but take a long time to compress, increasing your iteration times on release builds. Brotli **compression** is natively supported by Chrome and Firefox over https (see WebGL browser compatibility for more information).

Disabled: This disables compression. Use this if you want to implement your own compression in post-processing **scripts**.

Compression-built Unity builds work on any browser. Unity contains a software decompressor written in JavaScript, which it falls back to when compression on the http transfer level is not enabled by the server.

Advanced: Native browser decompression

The browser can handle decompression of Unity builds natively while it downloads the build data. This has the advantage of avoiding the additional delay caused by decompressing your files in JavaScript, therefore reducing your startup time. To let the browser handle decompression natively, you need to configure your web server to serve the compressed files with the appropriate http headers: These tell the browser that the data is compressed with gzip or Brotli so it decompresses the data while it is being transferred. Brotli compression is supported by Firefox and Chrome on https only, while gzip compression is supported by all browsers. See WebGL browser compatibility for more information.

Setting up web servers

The setup process for native browser decompression depends on your web server. The instructions on this page apply to the two most common web servers, **Apache** and **IIS**. Note that these are designed to work on a default setup, but may need adjustments to match your specific configuration. In particular, there may be issues if you already have other server-side configuration to compress hosted files, which may interfere with this setup. The basic idea is to append a *Content-Encoding* header to the server response, corresponding to the type of compression used at build time. This will allow the browser to perform decompression natively and asynchronously during download.

Apache

Apache server uses invisible *.htaccess* files for server configuration. The code below shows examples of *.htaccess* files which can be used to enforce native browser decompression. Note that Apache server configuration setup is optional.

For gzip-compressed builds put the following *.htaccess* file into your *Build* subfolder:

```
<IfModule mod_mime.c>
  AddEncoding gzip .unityweb
</IfModule>
```

For brotli-compressed builds put the following *.htaccess* file into your *Build* subfolder:

```
<IfModule mod_mime.c>
  AddEncoding br .unityweb
</IfModule>
```

IIS

Necessary IIS server configuration steps:

By default IIS server does not serve static content of unknown MIME type. In order to make Unity build work on IIS, you first need to associate the *.unityweb* extension with the *application/octet-stream* content typ. There are two ways to achieve that:

Using IIS Manager interface: Select your website in the IIS Manager panel, open the *MIME Types* feature and select *Add...* action. Set *.unityweb* as file name extension and *application/octet-stream* as MIME type, click *OK*.

Using server configuration file: IIS uses *web.config* files for server configuration. Those configuration files reflect all the changes made in IIS Manager for a specific folder. In order to associate *application/octet-stream* MIME type with *.unityweb* extension, you can use the following *web.config* file:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <system.webServer>
            <staticContent>
            <remove fileExtension=".unityweb" />
<mimeMap fileExtension=".unityweb" mimeType="application/octet-stream" />
            </staticContent>
    </system.webServer>
</configuration>
```

Note that configuration file affects all the server subfolders, therefore it is sufficient to set the MIME type for the *.unityweb* extension just once in the server root folder.

Optional IIS server configuration steps:

In order to speed up the build startup times, you may optionally use the following configuration files. Note that in order to use this setup, you need to have Microsoft's IIS URL Rewrite IIS module installed; otherwise, the browser will throw a 500 Internal Server Error.

For gzip-compressed builds put the following *web.config* file into the *Build* subfolder:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <system.webServer>
            <staticContent>
                    <remove fileExtension=".unityweb" />
                    <mimeMap fileExtension=".unityweb" mimeType="application/oct
            </staticContent>
            <rewrite>
                    <outboundRules>
                        <rule name="Append gzip Content-Encoding header">
                            <match serverVariable="RESPONSE_Content-Encoding" pa
                            <conditions>
                                    <add input="{REQUEST_FILENAME}" pattern="\.u
                            </conditions>
                            <action type="Rewrite" value="gzip" />
                        </rule>
                    </outboundRules>
            </rewrite>
    </system.webServer>
</configuration>
```

For brotli-compressed builds put the following *web.config* file into the *Build* subfolder:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <system.webServer>
            <staticContent>
                    <remove fileExtension=".unityweb" />
                    <mimeMap fileExtension=".unityweb" mimeType="application/oct
            </staticContent>
            <rewrite>
                    <outboundRules>
                        <rule name="Append br Content-Encoding header">
                            <match serverVariable="RESPONSE_Content-Encoding" pa
                            <conditions>
```

```
                                    <add input="{REQUEST_FILENAME}" pattern="\.u
                                </conditions>
                                <action type="Rewrite" value="br" />
                            </rule>
                        </outboundRules>
                </rewrite>
        </system.webServer>
</configuration>
```

Note that the `<remove fileExtension=".unityweb" />` lines are required to handle a situation when content type is already overridden at a higher level in the server directory's hierarchy, which could otherwise cause a server exception.

2017–05–24 Page amended with no editorial review

Updated in 5.6

Debugging and troubleshooting WebGL builds

Leave feedback

Visual Studio does not support debugging Unity **WebGL** content. To help you to find out exactly what's going on with your content, here are some tips on how to get information out of your build.

The browser's JavaScript console

Unity WebGL does not have access to your file system, so it does not write a log file like other platforms. However, it does write all logging information (such as `Debug.Log`, `Console.WriteLine` or Unity's internal logging) to the browser's JavaScript console.

To open the JavaScript console:

In Firefox, press Ctrl-Shift-K on Windows or Command-Option-K on a Mac.
In Chrome, press Ctrl-Shift-J on Windows or Command-Option-J on a Mac.
In Safari, go to Preferences > Advanced > Develop, and press Command-Option-C.
In Microsoft Edge or Internet Explorer, press F12.

Development builds

For debugging purposes, you might want to make a development build in Unity (open the Build Settings window and click the **Development Build** checkbox). Development builds allow you to connect the **profiler**, and Unity does not minify them, so the emitted JavaScript code still contains human-readable (though C++-mangled) function names. The browser can use these to display stack traces when you run into a browser error, when using `Debug.LogError, or when you throw an exception and exception support is disabled`. Unlike the managed stack traces that can occur when you have full exception support (see below), these stack traces have mangled names, and contain not only managed code, but also the internal UnityEngine code.

Exception support

WebGL has different levels of exception support (see documentation on Building for WebGL). By default, Unity WebGL only supports explicitly thrown exceptions. You can enable **Full** exception support, which emits additional checks in the IL2CPP-generated code, to catch access to null references and out-of-bounds array elements in your managed code. These additional checks significantly impact performance and increase code size and load times, so you should only use it for debugging.

Full exception support also emits function names to generate stack traces for your managed code. For this reason, stack traces appear in the console for uncaught exceptions and for `Debug.Log` statements. Use `System.Environment.Stacktrace` to get a stack trace string.

Troubleshooting

Problem: The build runs out of memory

This is a common problem, especially on 32-bit browsers. For more information on WebGL memory issues and how to fix them, see documentation on Memory in WebGL.

Problem: Files saved to Application.persistentDataPath do not persist

Unity WebGL stores all files that must persist between sessions (such as PlayerPrefs or files saved in persistentDataPath) to the browser IndexedDB. This is an asynchronous API, so you don't know when it's going to complete.

Call the following code to make sure Unity flushes all pending file system write operations to the IndexedDB file system from memory:

```
FS.syncfs(false, function (err) {
    console.log('Error: syncfs failed!');
 });
```

Error message: Incorrect header check

The browser console log usually prints this error as a result of incorrect server configuration. For more information on how to deploy a release build, see documentation on Deploying compressed builds.

Error message: Decompressing this format (1) is not supported on this platform

The browser console log prints this error when the content tries to load an AssetBundle compressed using LZMA, which Unity WebGL does not support. Re-compress the AssetBundle using LZ4 **compression** to solve this problem. For more information on **compression** for WebGL, see documentation on WebGL building, particularly the **AssetBundles** section.

2018–09–03 Page amended with limited editorial review

MonoDevelop replaced with Visual Studio from 2018.1

WebGL Graphics

Leave feedback

WebGL is an API for **rendering** graphics in web browsers, which is based on the functionality of the OpenGL ES graphics library. **WebGL** 1.0 roughly matches OpenGL ES 2.0 functionality, and **WebGL** 2.0 roughly matches OpenGL ES 3.0 functionality.

Deferred Rendering

Unity WebGL only supports Deferred Rendering Path if WebGL2.0 is available. On WebGL1.0, Unity WebGL runtime will fallback to Forward Rendering.

Global Illumination

Unity WebGL only supports baked GI. Realtime GI is not currently supported in WebGL. Furthermore, only Non-Directional **lightmaps** are supported.

Linear Rendering

Unity WebGL only supports linear color space rendering with WebGL 2.0. Linear color space rendering doesn't have fallback support for WebGL 1.0. To build a WebGL player using linear color space rendering, you need to remove WebGL 1.0 API in **Player Settings** > **Other Settings**; uncheck **Automatic Graphics API**.

Some web browsers don't support sRGB DXT texture compression. This can decrease the quality of rendering performance when using linear rendering, due to runtime decompression of all the DXT textures.

MovieTextures

WebGL does not support playing Video using the MovieTexture class. However, you can efficiently play back video in your WebGL content using the HTML5 video element. Download this Asset Store package for an example of how to do so.

WebGL Shader code restrictions

The WebGL 1.0 specification imposes some limitations on GLSLS **shader** code, which are more restricted then what many OpenGL ES 2.0 implementations allow. This is mostly relevant when you write your own shaders.

Specifically, WebGL has restriction on which values can be used to index arrays or matrices: WebGL only allows dynamic indexing with constant expressions, loop indices or a combination. The only exception is for uniform access in **vertex shaders** , which can be indexed using any expression.

Also, restrictions apply on control structures. The only type of loops which are allowed are counting *for* loops, where the initializer initializes a variable to a constant, the update adds a constant to or subtracts a constant from the variable, and the continuation test compares the variable to a constant. *for* loops which don't match those criteria and *while* loops are not allowed.

Font rendering

Unity WebGL supports dynamic font rendering like all Unity platforms. However, it does not have access to the fonts installed on the user's machine, so any fonts used must be included in the project folder (including any fallback fonts for international characters, or bold/italic versions of fonts), and set up as fallback font names.

Anti-Aliasing

WebGL supports anti-aliasing on most (but not on all) combinations of browsers and GPUs. To use it, anti-aliasing must be enabled in the default Quality Setting for the WebGL platform.

Note that on WebGL1.0 there are several limitations:

Switching quality settings at runtime will not enable or disable anti-aliasing - it has to be set up in the default Quality Setting loaded at player start up.

The different multi sampling levels (2x, 4x, etc..) have no effect in WebGL, it's either on or off.

Any post-processing effect applied to the **camera** disables the built-in Anti-Aliasing.

HDR is not compatible with anti-aliasing so make sure "Allow **HDR**" camera option is disabled.

There are no such limitations on WebGL2.0.

Reflection Probes

Reflection probes are supported in WebGL, but due to limitations in the WebGL specification about rendering to specific mipmaps, smooth realtime **reflection probes** are not supported (so realtime reflection probes will always generate sharp reflections, which may appear very low-resolution). Smooth realtime reflection probes will require WebGL 2.0.

WebGL 2.0 support

Unity includes support for the WebGL 2.0 API, which brings OpenGL ES 3.0-level rendering capabilities to the web.

By default, Unity WebGL builds support both WebGL 1.0 and WebGL 2.0 APIs, This can be configured in the WebGL **Player Settings** > **Other Settings**; to do this, uncheck **Automatic Graphics API**.

When WebGL 2.0 is supported in browsers, content can benefit from a better quality in the **Standard Shader**, GPU Instancing support, directional lightmap support, no restrictions on indexing and loops in shader code, and better performance.

2017–06–19 Page amended with limited editorial review

Linear rendering for WebGL 2.0 added in 2017.2

WebGL Networking

Leave feedback

No direct socket access

Due to security implications, JavaScript code does not have direct access to IP Sockets to implement network connectivity. As a result, the .NET **networking** classes (ie, everything in the `System.Net` namespace, particularly `System.Net.Sockets`) are non-functional in **WebGL**. The same applies to Unity's old `UnityEngine.Network*` classes, which are not available when building for WebGL.

If you need to use Networking in WebGL, you currently have the options to use the `WWW` or `UnityWebRequest` classes in Unity or the new Unity Networking features which support WebGL, or to implement your own networking using WebSockets or WebRTC in JavaScript.

Using the WWW or WebRequest classes in WebGL

The **WWW** and **UnityWebRequest** classes are supported in WebGL. They are implemented using the `XMLHttpRequest` class in JavaScript, using the browser to handle WWW requests. This imposes some security restrictions on accessing cross-domain resources. Basically any WWW request to a server which is different from the server hosting the WebGL content needs to be authorized by the server you are trying to access. To access cross-domain WWW resources in WebGL, the server you are trying to access needs to authorize this using CORS.

If you try to access content using `WWW` or `UnityWebReqest`, and the remote server does not have CORS set up or configured correctly, you will see an error like this in the browser console:

```
Cross-Origin Request Blocked: The Same Origin Policy disallows reading the remot
```

CORS stands for Cross-Origin Resource Sharing, and is documented here. Basically, the server needs to add some `Access-Control` headers to the http responses it sends out, which will tell browsers that it is allowed to let web pages access the content on the server. Here's an example of a header setup, which would allow Unity WebGL to access resources on a web server from any origin, with common request headers and using the http `GET`, `POST` or `OPTIONS` methods:

```
"Access-Control-Allow-Credentials": "true",
"Access-Control-Allow-Headers": "Accept, X-Access-Token, X-Application-Name, X-R
"Access-Control-Allow-Methods": "GET, POST, OPTIONS",
"Access-Control-Allow-Origin": "*",
```

Note that WWW.responseHeaders is limited to a subset of the actual response headers, according to 7.1.1 of the CORS specification.

Also note that XMLHttpRequest does not allow streaming of data, thus the WWW class on WebGL will only process any data once the download has finished (so AssestBundles cannot be decompressed and loaded while downloading as on other platforms).

Do not block on WWW or WebRequest downloads

Do not use code which blocks on a WWW or WebRequest download, like this:

```
while(!www.isDone) {}
```

Blocking on WWW or WebRequest downloads does not work on Unity WebGL. Because WebGL is single threaded, and because the `XMLHttpRequest` class in JavaScript is asynchronous, your download never finishes unless you return control to the browser; instead, your content deadlocks. Instead, use a Coroutine and a `yield` statement to wait for the download to finish.

Using Unity Networking

Unity Networking supports WebGL by enabling communication via the WebSockets protocol. See **Networking.NetworkManager.useWebSockets** .

Using Web Sockets or WebRTC from JavaScript

As written above, direct access to IP Sockets is not possible in WebGL. However, if you need networking capabilities beyond the WWW class, it is possible to use Web Sockets or WebRTC, both networking protocols supported by browsers. Web Sockets has wider support, but WebRTC allows p2p connections between browsers and unreliable connections. Neither of these protocols are exposed through built-in APIs in Unity yet, but it is possible to use a JavaScript plugin to implement this. A sample of a **plugin** which implements WebSocket networking in Unity WebGL can be found on AssetStore.

Using Audio In WebGL

Leave feedback

Audio in **WebGL** is done differently then on all other platforms. On other platforms we use FMOD internally to supply audio playback and mixing. Since the **WebGL** platform does not support threads, we need to use a different implementation, which is internally based on the Web Audio API, which lets the browser handle audio playback and mixing for us.

Unfortunately, this limits audio functionality in Unity WebGL to supporting only the most basic features. This page will document what is expected to work. Anything not listed here is not currently supported on WebGL.

AudioSource

Audio sources support basic positional audio playback with pausing and resuming, panning, rolloff, pitch setting, and doppler effect support.

The following `AudioSource` APIs are supported:

Properties

`clip`
`dopplerLevel`
`ignoreListenerPause`
`ignoreListenerVolume`
`isPlaying`
`loop`
`maxDistance`
`minDistance`
`mute`
`pitch` (Note that only positive values for pitch are supported.)
`playOnAwake`
`rolloffMode`
`time`
`timeSamples`
`velocityUpdateMode`
`volume`

Methods

`Pause`
`Play`
`PlayDelayed`
`PlayOneShot`
`PlayScheduled`
`SetScheduledEndTime`
`SetScheduledStartTime`
`Stop`
`UnPause`

`PlayClipAtPoint`

AudioListener

All `AudioListener` APIs are supported.

AudioClip

Audio clips in WebGL will always be imported in the AAC format, as that is widely supported by different browsers.

The following `AudioClip` APIs are supported:

Properties

`length`
`loadState`
`samples`

Methods

`AudioClip.Create` is supported partially: it only works if the streaming parameter is set to false and the complete audio samples can be loaded at the time `AudioClip.Create` is called. It then creates the clip and loads all samples before returning control.
`AudioClip.SetData` is supported partially: it only works for replacing the entire contents of the AudioClip. The `offsetSamples` parameter is ignored.

SystemInfo.supportsAudio

`SystemInfo.supportsAudio` is not implemented on WebGL and always returns true.

WWW.audioClip

`WWW.audioClip` should work in WebGL, if the **audio clip** is in a format which is natively supported by the browser. See Mozilla's documentation on supported media formats for a list of supported formats in different browsers.

Microphone

The `Microphone` class is not supported in WebGL.

WebGL performance considerations

Leave feedback

What kind of performance can you expect on WebGL?

This is a bit difficult to answer, as it depends on many factors.

In general, you can assume that you will get performance close to native apps for the GPU side, as the **WebGL** graphics API uses your GPU for hardware accelerated **rendering** - there is just a little overhead for translating **WebGL** API calls and **shaders** to your OS graphics API (typically DirectX on Windows or OpenGL on Mac or Linux).

For the CPU side, all your code is translated into asm.js JavaScript. So what kind of performance you can expect depends a lot on the JavaScript engine of the web browser used, and there are some pretty significant differences there currently. At the point of this writing (November 2015), Microsoft Edge and Mozilla Firefox deliver the best performance on Unity code, as these are currently the only browsers which make use of the asm.js spec to use an optimized **AOT compilation** path of JavaScript code for that case, which delivers performance within a factor of less then 2x slowdown compared to native code for many programming benchmarks - and that factor also matches results we've seen from different unity content we deployed to WebGL and ran in Firefox and Edge.

There are some other considerations, though. Currently, the JavaScript language does neither support multi-threading, nor SIMD. So, any code which benefits from these features will see bigger slowdowns then other code. You cannot write threading or SIMD code in WebGL in your **scripts**, but some engine parts are normally multi-threaded or SIMD optimized, and will run less performant on WebGL because of this. An example is the **skinning** code, which is both multi-threaded and SIMD-optimized. You can use the new **timeline** profiler in Unity to see how Unity distributes work to different threads on non-WebGL platforms. Longer term, we are hoping that these features will become available on WebGL as well.

WebGL-specific settings which affect performance

For best performance, set the optimization level to "Fastest" in the Build Player dialog, and set "Exception support" to "None" in the **player settings** for WebGL.

Profiling WebGL

The Unity profiler is supported in WebGL, see here how to set it up.

WebGL content in background tabs

If the *Run in background* is enabled in the WebGL Player Settings, or if you enable **Application.runInBackground** , then your content will continue to run when the canvas or the browser window loses focus.

However, it should be noted that browsers may throttle content running in background tabs. If the tab with your content is not visible, your content will only be updated once a second in most browsers. Note that this will cause **Time.time** to progress slower than usual with the default settings, as the default value of **Time.maximumDeltaTime** is lower than one second.

Throttling WebGL performance

You may want to run your WebGL content at a lower frame rate in some situations to reduce CPU usage. Like on other platforms, you can use the Application.targetFrameRate API to do so.

When you don't want to throttle performance, set this API to the default value of –1, rather then to a high value. This allows the browser to adjust the frame rate for the smoothest animation in the browser's render loop, and may produce better results than Unity trying to do its own main loop timing to match a target frame rate.

Memory in WebGL

Leave feedback

Memory in Unity **WebGL** can be a constraining factor restricting the complexity of the content you can run, so we would like to provide some explanation on how memory is used in **WebGL**.

Your WebGL content will run inside a browser, so any memory has to be allocated by the browser within the browser's memory space. The amount of available memory can vary a lot depending on the browser, OS and device used. Determining factors include whether the browser is a 32 or 64 bit process, whether the browser uses separate processes for each tab or has your content share a memory space with all other open tabs, and how much memory the browser's JavaScript engine requires to parse your code.

There are multiple areas where Unity WebGL content will require the browser to allocate significant amounts of memory:

Unity Heap

This is the memory Unity uses to store all it's state, managed and native objects and currently loaded **assets** and **scenes**. This is similar to the memory used by Unity players on any other platform. You can configure the size of this in the Unity WebGL **player settings** (But for faster iteration, you can also edit the TOTAL_MEMORY value written to the generated html file). You can use the Unity **Profiler** to profile and sample the contents of this memory. This memory will be created as a TypedArray of bytes in JavaScript code, and requires the browser be able to allocate a consecutive block of memory of this size. You will want this space to be as small as possible (so that the browser will be able to allocate it even if memory is fragmented), but large enough to fit all the data required to play any scene of your content.

Asset Data

When you create a Unity WebGL build, Unity will write out a .data file containing all the scenes and assets needed for your content. Since WebGL does not have a real file system, this file will be downloaded before your content can start, and the uncompressed data will be kept in a consecutive block of browser memory for the whole time your content is run. So, to keep both download times and memory usage low, you should try to keep this data as small as possible. See the documentation page on Reducing File size for information on how to optimize the build size of your assets.

Another thing you can do to reduce load times and the amount of memory used for assets is to pack your asset data into AssetBundles. By doing so, you get full control of when your assets need to be downloaded, and you can unload them when you no longer need them, which will free any memory used by them. Note that AssetBundles will be loaded directly into the Unity heap and will not result in additional allocations by the browser (unless you use Asset Bundle Caching using **WWW.LoadFromCacheOrDownload** , which is using a memory-mapped Virtual File System, backed by the browser's IndexedDB).

Memory needed to parse the code

Another issue related to memory is the memory required by the browser's JavaScript engine. Unity will emit very large files of millions of lines of generated JavaScript code, which is an order of magnitude larger than common uses of JavaScript code in browsers. Some JavaScript engines may allocate some rather large data structures to parse and optimize this code, which may results in memory spikes of up to several Gigabytes of memory when loading the content in some cases. We expect that future technologies like WebAssembly will eventually eliminate this problem, but until then, the best advise we can give is to do what you can to keep the size of the emitted code low. See the comments on distribution size here for more information on how to do that.

Dealing with memory issues

When you see an error related to memory in a Unity WebGL build, it is important to understand whether it is the browser which is failing to allocate memory or if the Unity WebGL runtime is failing to allocate a free block of memory within the pre-allocated block of the Unity heap. If the browser is failing to allocate memory, then it may help to try to reduce the size used by one or more of the memory areas above (for instance by reducing the size of the Unity heap). On the other hand, if the Unity runtime is failing to allocate a block inside the Unity heap, you may want to increase the size of that instead.

Unity will try to interpret error messages to tell which of the two it is (and provide suggestions on what to do). Since different browsers may report different messages, that is not always easy, however, and we may not be interpreting all of them. When you

see a generic “Out of memory” error from the browser, it is likely to be an issue of the browser running out of memory (where you might want to use a smaller Unity heap). Also, you may sometimes see browsers simply crashing when loading Unity content without showing a human-parseable error message. This can have many reasons, but is frequently caused by JavaScript engines requiring too much memory to parse and optimize the generated code.

Large-Allocation Http Header

Your server can emit the Large-Allocation http header for your content. This tells supported browsers (currently only Firefox) about your memory needs, allowing them to spawn a new process with an unfragmented memory space, or to perform other housekeeping to make sure that the large allocation succeeds. This can solve issues where the browser runs out of memory when trying to allocate the Unity heap, especially on 32-bit browsers.

Garbage Collection considerations

When you allocate managed objects in Unity, they will need to be garbage collected when they are no longer used. See our documentation on automatic memory management for more information. In WebGL, this is the same. Managed, garbage collected memory is allocated inside the Unity heap.

One distinction in WebGL, however, concerns the points in time when garbage collection (GC) can take place. To perform garbage collection, the GC would normally need to pause all running threads and inspect their stacks and registers for loaded object references. This is not currently possible in JavaScript. For this reason, the GC will only run in WebGL in situations where the stack is known to be empty (which is currently once after every frame). This is not a problem for most content which deals with managed memory conservatively and has relatively few GC allocations within each frame (you can debug this using the Unity profiler).

However, the following code would fail running on WebGL, becuase it would not get a chance to run the GC between iterations of the loop, to free up memory used by all the intermediate string objects - which would eventually cause it to run out of memory in the Unity heap.

```
string hugeString = "";

for (int i = 0; i < 100000; i++)
{
    hugeString += "foo";
}
```

Further reading

Unity blog: Understanding Memory in Unity WebGL

Unity blog: Unity WebGL Memory: The Unity Heap

2018–08–23 Page amended with no editorial review

WebGL: Interacting with browser scripting

Leave feedback

When building content for the web, you might need to communicate with other elements on your web page. Or you might want to implement functionality using Web APIs which Unity does not currently expose by default. In both cases, you need to directly interface with the browser's JavaScript engine. Unity **WebGL** provides different methods to do this.

Calling JavaScript functions from Unity scripts

The recommended way of using browser JavaScript in your project is to add your JavaScript sources to your project, and then call those functions directly from your script code. To do so, place files with JavaScript code using the .jslib extension under a "Plugins" subfolder in your **Assets** folder. The **plugin** file needs to have a syntax like this:

```
mergeInto(LibraryManager.library, {

  Hello: function () {
    window.alert("Hello, world!");
  },

  HelloString: function (str) {
    window.alert(Pointer_stringify(str));
  },

  PrintFloatArray: function (array, size) {
    for(var i = 0; i < size; i++)
    console.log(HEAPF32[(array >> 2) + i]);
  },

  AddNumbers: function (x, y) {
    return x + y;
  },

  StringReturnValueFunction: function () {
    var returnStr = "bla";
    var bufferSize = lengthBytesUTF8(returnStr) + 1;
    var buffer = _malloc(bufferSize);
    stringToUTF8(returnStr, buffer, bufferSize);
    return buffer;
  },

  BindWebGLTexture: function (texture) {
    GLctx.bindTexture(GLctx.TEXTURE_2D, GL.textures[texture]);
  },
```

```
});
```

Then you can call these functions from your C# **scripts** like this:

```
using UnityEngine;
using System.Runtime.InteropServices;

public class NewBehaviourScript : MonoBehaviour {

    [DllImport("__Internal")]
    private static extern void Hello();

    [DllImport("__Internal")]
    private static extern void HelloString(string str);

    [DllImport("__Internal")]
    private static extern void PrintFloatArray(float[] array, int size);

    [DllImport("__Internal")]
    private static extern int AddNumbers(int x, int y);

    [DllImport("__Internal")]
    private static extern string StringReturnValueFunction();

    [DllImport("__Internal")]
    private static extern void BindWebGLTexture(int texture);

    void Start() {
        Hello();

        HelloString("This is a string.");

        float[] myArray = new float[10];
        PrintFloatArray(myArray, myArray.Length);

        int result = AddNumbers(5, 7);
        Debug.Log(result);

        Debug.Log(StringReturnValueFunction());

        var texture = new Texture2D(0, 0, TextureFormat.ARGB32, false);
        BindWebGLTexture(texture.GetNativeTextureID());
```

```
    }
}
```

Simple numeric types can be passed to JavaScript in function parameters without requiring any conversion. Other data types will be passed as a pointer in the emscripten heap (which is really just a big array in JavaScript). For strings, you can use the `Pointer_stringify` helper function to convert to a JavaScript string. To return a string value you need to call `_malloc` to allocate some memory and the `stringToUTF8` helper function to write a JavaScript string to it. If the string is a return value, then the **il2cpp** runtime will take care of freeing the memory for you. For arrays of primitive types, `emscripten` provides different `ArrayBufferViews` into it's heap for different sizes of integer, unsigned integer or floating point representations of memory: **HEAP8, HEAPU8, HEAP16, HEAPU16, HEAP32, HEAPU32, HEAPF32, HEAPF64**. To access a texture in WebGL, emscripten provides the `GL.textures` array which maps native texture IDs from Unity to WebGL texture objects. WebGL functions can be called on emscripten's WebGL context, `GLctx`.

For more information on how to interact with JavaScript, refer to the emscripten documentation.

In addition, note that in the Unity installation folder there are several plugins that you can use as reference, in `PlaybackEngines/WebGLSupport/BuildTools/lib` and `PlaybackEngines/WebGLSupport/BuildTools/Emscripten/src/library*`.

Legacy ways of calling JavaScript code from Unity

Note: Starting from Unity 5.6 the recommended way of calling JavaScript code from Unity is through a .jslib plugin. The method described below is only supported for compatibility reasons and might become deprecated in the future versions of Unity.

You can use the Application.ExternalCall() and Application.ExternalEval() functions to invoke JavaScript code on the embedding web page. Note that expressions are evaluated in the local scope of the build. If you would like to execute JavaScript code in the global scope, see the *Code Visibility* section below.

Calling Unity scripts functions from JavaScript

Sometimes you need to send some data or notification to the Unity script from the browser's JavaScript. The recommended way of doing it is to call methods on **GameObjects** in your content. If you are making the call from a JavaScript plugin, embedded in your project, you can use the following code:

```
SendMessage(objectName, methodName, value);
```

Where **objectName** is the name of an object in your scene; **methodName** is the name of a method in the script, currently attached to that object; **value** can be a string, a number, or can be empty. For example:

```
SendMessage('MyGameObject', 'MyFunction');
SendMessage('MyGameObject', 'MyFunction', 5);

SendMessage('MyGameObject', 'MyFunction', 'MyString');
```

If you would like to make a call from the global scope of the embedding page, see the *Code Visibility* section below.

Calling C functions from Unity scripts

Unity compiles your sources into JavaScript from C/C++ code using emscripten, so you can also write plugins in C/C++ code, and call these functions from C#. So, instead of the jslib file in the example above, you could have a C/C++ file in your project - it will automatically get compiled with your scripts, and you can call functions from it, just like in the JavaScript example above.

If you are using C++ (.cpp) to implement the plugin then you must ensure the functions are declared with C linkage to avoid name mangling issues:

```
#include <stdio.h>

extern "C" void Hello ()
{
    printf("Hello, world!\n");
}

extern "C" int AddNumbers (int x, int y)
{
    return x + y;
}
```

Code visibility

Starting from Unity 5.6 all the build code is executed in its own scope. This approach makes it possible to embed your game on an arbitrary page without causing conflicts with the embedding page code, as well as makes it possible to embed more than one build on the same page.

If you have all your JavaScript code in the form of *.jslib* plugins inside your project, then this JavaScript code will run inside the same scope as the compiled build and your code should work pretty much the same way as in previous versions of Unity (for example, the following objects and functions should be directly visible from the JavaScript plugin code: *Module, SendMessage, HEAP8, ccall etc.*).

However, if you are planning to call the internal JavaScript functions from the global scope of the embedding page, you should always assume that there are multiple builds embedded on the page, so you should explicitly specify which build you are referencing to. For example, if your game has been instantiated as:

```
var gameInstance = UnityLoader.instantiate("gameContainer", "Build/build.json",
{onProgress: UnityProgress});
```

Then you can send a message to the build using *gameInstance.SendMessage()*, or access the build Module object using *gameInstance.Module*.

2017–11–14 Page amended with no editorial review

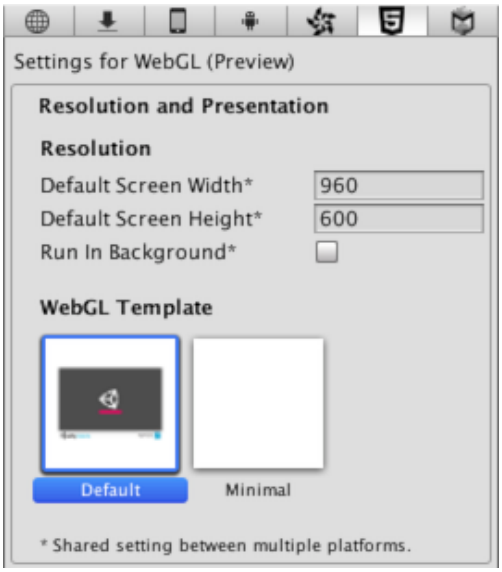
Fixed error in code example.

Updated in 5.6

Using WebGL Templates

Leave feedback

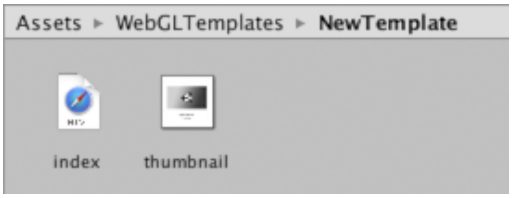
When you build a WebGL project, Unity embeds the player in an HTML page so that it can be played in the browser. The default page is a simple white page with a loading bar on a grey canvas. Alternatively, you can select a minimal template (with only the necessary boilerplate code to run the WebGL content) in the Player Settings inspector (menu: Edit > Project Settings > Player).



The built-in HTML pages are fine for testing and demonstrating a minimal player but for production purposes, it is often desirable to see the player hosted in the page where it will eventually be deployed. For example, if the Unity content interacts with other elements in the page via the external call interface then it must be tested with a page that provides those interacting elements. Unity allows you to supply your own pages to host the player by using **WebGL templates**.

Structure of a WebGL Template

Custom templates are added to a project by creating a folder called "WebGLTemplates" in the **Assets** folder - the templates themselves are sub-folders within this folder. Each template folder contains an *index.html* file along with any other resources the page needs, such as images or stylesheets.



Once created, the template will appear among the options on the **Player Settings inspector**. (the name of the template will be the same as its folder). Optionally, the folder can contain a file named *thumbnail.png*, which should have dimensions of 128x128 **pixels**. The thumbnail image will be displayed in the inspector to hint at what the finished page will look like.

The html file needs to contain at least the following elements:

Script tag for the Unity **WebGL** loader: `<script src="%UNITY_WEBGL_LOADER_URL%"></script>`

Script for instantiating the game: `<script> var gameInstance = UnityLoader.instantiate("gameContainer", "%UNITY_WEBGL_BUILD_URL%");</script>`

A `<div>` tag, which *id* is used in the instantiation function. The contents of this div will be replaced with the game instance.

UnityLoader.instantiate(container, url, override)

UnityLoader.instantiate is responsible for creating a new instance of your content.

container can be either a DOM element (normally a `<div>` element) or an id of a DOM element. If the DOM element is provided, then the game will be instantiated immediately. If an id of a DOM element is provided, then the game will be instantiated after the whole document is parsed (which means you can provide an id of a DOM element which has not yet been created at the time of *UnityLoader.instantiate()* call).

url specifies the address of the json file, which contains information about the build (you may use the *%UNITY_WEBGL_BUILD_URL%* variable which will be automatically resolved at build time).

override is an optional parameter which can be used to override the default properties of the game instance. For example, you can override the *compatibilityCheck*, *onProgress* and *popup* functions, as those are properties of the game instance. Note that *Module* is a property of the game instance as well, so the properties of the *Module* can be overridden at instantiation time too. Consider the following example:

```
UnityLoader.instantiate("MyContainer", "%UNITY_WEBGL_BUILD_URL%", {
  compatibilityCheck: function (gameInstance, onsuccess, onerror) {
    if (!UnityLoader.SystemInfo.hasWebGL) {
      gameInstance.popup("Your browser does not support WebGL",
        [{text: "OK", callback: onerror}]);
    } else if (UnityLoader.SystemInfo.mobile) {
      gameInstance.popup("Please note that Unity WebGL is not currently supported on mo
        [{text: "OK", callback: onsuccess}]);
    } else if (["Edge", "Firefox", "Chrome", "Safari"].indexOf(UnityLoader.SystemInfo.b
      gameInstance.popup("Please note that your browser is not currently supported for
        [{text: "OK", callback: onsuccess}]);
    } else {
      onsuccess();
    }
  },
  onProgress: MyProgressFunction,
  Module: {
    onRuntimeInitialized: MyInitializationCallbackFunction,
  },
});
```

Template tags

During the build process, Unity looks for special tag strings in the page text and replaces them with values supplied by the Editor. These include the name, onscreen dimensions and other useful information about the player.

Tags are delimited by percent signs (%) in the page source. For example, if the product name is defined as **MyPlayer** in the Player settings:-

```
<title>%UNITY_WEB_NAME%</title>
```

...in the template's index file will be replaced with

```
<title>MyPlayer</title>
```

...in the host page generated for the build. The complete set of tags is given below:-

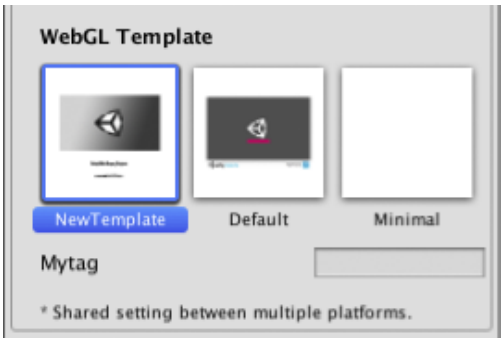
UNITY_WEB_NAME: Name of the player.

UNITY_WEBGL_LOADER_URL: URL of the UnityLoader.js script, which performs instantiation of the build.

UNITY_WEBGL_BUILD_URL: URL of the JSON file, containing all the necessary information about the build.

UNITY_WIDTH and **UNITY_HEIGHT**: Onscreen width and height of the player in pixels.

UNITY_CUSTOM_SOME_TAG: If you add a tag to the index file in the form `UNITY_CUSTOM_XXX`, then this tag will appear in the Player Settings when your template is selected. For example, if you add `<title>Unity Player | %UNITY_CUSTOM_MYTAG%</title>` to the source, the Player Settings will look like this:-



The textbox next to the tag's name contains the text that the custom tag will be replaced with during the build.

Example

To illustrate the use of the template tags, here is the HTML source that Unity uses for its minimal WebGL template.

```
<!DOCTYPE html>
<html lang="en-us">

  <head>
    <meta charset="utf-8">
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>Unity WebGL Player | %UNITY_WEB_NAME%</title>
    <script src="%UNITY_WEBGL_LOADER_URL%"></script>
    <script>
    var gameInstance = UnityLoader.instantiate("gameContainer", "%UNITY_WEBGL_BUILD_URL
    </script>
  </head>

  <body>
    <div id="gameContainer" style="width: %UNITY_WIDTH%px; height: %UNITY_HEIGHT%px; ma
  </body>

</html>
```

Both minimal and default templates can be found in the Unity installation folder under Editor\Data\PlaybackEngines\WebGLSupport\BuildTools\WebGLTemplates on Windows or /PlaybackEngines/WebGLSupport/BuildTools/WebGLTemplates on Mac.

Adding a progress bar

Unity WebGL content will automatically render a default progress bar for you when it loads. You can override the default loading bar by providing your own progress function as an additional instantiation parameter. For example:

```
var gameInstance = UnityLoader.instantiate("gameContainer", "%UNITY_WEBGL_BUILD_URL%",
```

where `UnityProgress` is a function of 2 arguments: `gameInstance` (identifies the game instance the progressbar belongs to) and `progress` (a value from 0.0 to 1.0, providing information about the current loading progress).

For example, the progress function in the default WebGL template looks the following way:

```
var gameInstance = UnityLoader.instantiate("gameContainer", "%UNITY_WEBGL_BUILD_URL%",
```

For example, the progress function in the default WebGL template looks the following way:

```
function UnityProgress(gameInstance, progress) {
  if (!gameInstance.Module)
    return;
  if (!gameInstance.logo) {
    gameInstance.logo = document.createElement("div");
    gameInstance.logo.className = "logo " + gameInstance.Module.splashScreenStyle;
    gameInstance.container.appendChild(gameInstance.logo);
  }
  if (!gameInstance.progress) {
    gameInstance.progress = document.createElement("div");
    gameInstance.progress.className = "progress " + gameInstance.Module.splashScreenSty
    gameInstance.progress.empty = document.createElement("div");
    gameInstance.progress.empty.className = "empty";
    gameInstance.progress.appendChild(gameInstance.progress.empty);
    gameInstance.progress.full = document.createElement("div");
    gameInstance.progress.full.className = "full";
    gameInstance.progress.appendChild(gameInstance.progress.full);
    gameInstance.container.appendChild(gameInstance.progress);
  }
  gameInstance.progress.full.style.width = (100 * progress) + "%";
  gameInstance.progress.empty.style.width = (100 * (1 - progress)) + "%";
  if (progress == 1)
    gameInstance.logo.style.display = gameInstance.progress.style.display = "none";
}
```

You can use it as is or as reference for your own templates. Since the progress bar is completely implemented in JavaScript, you can customize or replace it to show anything you want as a progress indication.

2017–05–24 Page amended with no editorial review

Updated in 5.6

Cursor locking and full-screen mode in WebGL

Leave feedback

Cursor locking (using **Cursor.lockState**) and full-screen mode (using **Screen.fullScreen**) are both supported in Unity **WebGL**, implemented using the respective HTML5 APIs (Element.requestPointerLock and Element.requestFullscreen). These are supported in Firefox and Chrome. Safari cannot currently use full-screen and cursor locking.

Enabling cursor locking and full-screen mode in WebGL

Due to security concerns, browsers will only allow locking the cursor or going into full-screen mode in direct response to a user-initiated event (like a mouse click or key press). Unfortunately, Unity does not have separate event and **rendering** loops, so it defers event handling to a point where the browser no longer acknowledges a full-screen or cursor lock request issued from Unity scripting as a direct response to the event which triggered it. As a result, Unity triggers the request on the next user-initiated event, rather than the event that triggered the cursor lock or full-scree request.

To make this work with acceptable results, you should trigger cursor locking or full-screen requests on mouse/key down events, instead of mouse/key up events. This ensures that when the request is deferred to the next user-initiated event, it is triggered when the user releases the mouse or key.

If you use Unity's UI.Button **component**, you can achieve the desired behaviour by creating a subclass of `Button`, which overrides the OnPointerDown method.

Note that browsers may show a notification message or ask the user for permission before entering full-screen mode or locking the cursor.

Input in WebGL

Leave feedback

Gamepad and Joystick support

Gamepads and Joysticks are supported in **WebGL** (using the **Input** class) on browsers which support the HTML5 Gamepad API. Check our browser compatibilty table to see which browsers these are.

Note that browsers may only allow access to available input devices once the user has interacted with the device while the content has focus. This is a security measure to prevent using the connected devices for browser fingerprinting purposes. For this reason, you should make sure to instruct the user to click a button on their devices before checking **Input.GetJoystickNames()**) for connected devices.

Touch support

While Unity WebGL does not officially support mobile devices yet, **Input.touches** and related APIs are implemented on browsers and devices with touch support - as well as **Input.acceleration** .

Keyboard input and focus handling

By default, Unity WebGL will process all keyboard input send to the page, regardless of whether the WebGL canvas has focus or not. This is done so that a user can start playing a keyboard-based game right away without the need to click on the canvas to focus it first. However, this can cause problems when there are other HTML elements on the page which should receive keyboard input, such as text fields - as Unity will consume the input events before the rest of the page can get them. If you need to have other HTML elements receive keyboard input, you can change this behavior using the **WebGLInput.captureAllKeyboardInput** property.

iOS

Leave feedback

This section of the User Manual contains documentation on developing for the **iOS** platform. Use the navigation column on the left-hand side of the page to view topics in this section.

Getting started with iOS development

Leave feedback

Building games for devices like the iPhone and iPad requires a different approach than you would use for desktop PC games. Unlike the PC market, your target hardware is standardized and not as fast or powerful as a computer with a dedicated video card. Because of this, you will have to approach the development of your games for these platforms a little differently. Also, the features available in Unity for **iOS** differ slightly from those for desktop PCs.

Setting up your Apple Developer account

You don't need an Apple Developer account to build to devices; any Apple ID is sufficient for building only to your own device for testing.

However, we recommend that you set up your Apple Developer account before proceeding because you will need it to use Unity to its full potential with iOS. This includes establishing your team, adding your devices, and finalizing your provisioning profiles. All this setup is performed through Apple's Developer website. Since this is a complex process, we have provided a basic outline of the tasks that must be completed, which can be referred to alongside the step-by-step instructions at Apple's iPhone Developer portal.

The Unity XCode project

When you build the Unity iOS game an XCode project is generated. This project is required to sign, compile and prepare your game for distribution. See the Unity XCode project manual page for further information.

Accessing iOS functionality

Unity provides a number of scripting APIs to access the multi-touch screen, accelerometer, device geographical location system and much more. You can find out more about the script classes on the iOS scripting page.

Exposing native C, C++ or Objective-C code to scripts

Unity allows you to call custom native functions written in C, C++ or Objective-C directly from C# **scripts**. To find out how to bind native functions, visit the plugins page.

Prepare your application for in-app purchases

The Unity iOS runtime allows you to download new content and you can use this feature to implement in-app purchases. See the downloadable content manual page for further information.

Splash screen customization

See the splash screen customization page to find out how to change the image your game shows while launching.

Troubleshooting and reporting crashes

If you are experiencing crashes on the iOS device, please consult the iOS troubleshooting page for a list of common issues and solutions. If you can't find a solution here then please file a bug report for the crash (menu: **Help > Report A Bug** in the Unity editor).

How Unity's iOS and desktop targets differ

Audio compression

Unity supports importing a variety of source format sound files. However when importing these files (with the exception of tracker files), they are always re-encoded to the build target format. By default, this format is Vorbis, though this can be

overridden per platform to other formats (ADPCM, MP3 etc) if required. MM3 playback offers slightly better performance on iPhone compared with Vorbis playback.

PVRTC instead of DXT texture compression

Unity iOS does not support DXT textures. Instead, **PVRTC** texture **compression** is natively supported by iPhone/iPad devices. Consult the texture import settings documentation to learn more about iOS **texture formats**.

Movie/Video playback

We recommend you use the Video Player to play video files. This supersedes the earlier Movie Texture feature.

2018–06–14 Page amended with limited editorial review

Video Player component added in Unity 5.6

Unity iOS Basics

Leave feedback

This section covers the most common and important questions that come up when starting to work with **iOS**.

Prerequisites

I've just received iPhone Developer approval from Apple, but I've never developed for iOS before. What do I do first?

A: Download the SDK, get up and running on the Apple developer site, and set up your team, devices, and provisioning. We've provided a basic list of steps to get you started.

Can Unity-built games run in the iPhone Simulator?

A: No, but Unity iOS can build to iPad Simulator if you're using the latest SDK. However the simulator itself is not very useful for Unity because it does not simulate all inputs from iOS or properly emulate the performance you get on the iPhone/iPad. You should test out gameplay directly inside Unity using the iPhone/iPad as a remote control while it is running the Unity Remote application. Then, when you are ready to test performance and optimize the game, you should publish to iOS devices.

Unity Features

How do I work with the touch screen and accelerometer?

A: In the scripting reference inside your Unity iOS installation, you will find classes that provide the hooks into the device functionality that you will need to build your apps. Consult the Input page for more information.

My existing particle systems seem to run very slowly on iOS. What should I do?

A: iOS has a relatively low fillrate. If your particles cover a large portion of the screen with multiple layers, it will kill iOS performance even with the simplest **shader**. We suggest baking your particle effects into a series of textures offline. Then, at run-time, you can use 1–2 particles to display them via animated textures. You can get ok results with a minimum amount of overdraw this way.

Can I make a game that uses heavy physics?

A: Physics can be expensive on iOS as it requires a lot of floating point number calculations. You should completely avoid MeshColliders if at all possible, but they can be used if they are really necessary. To improve performance, use a low fixed framerate using **Edit->Time->Fixed Delta Time**. A framerate of 10–30 is recommended. Enable rigidbody interpolation to achieve smooth motion while using low physics frame rates. In order to achieve completely fluid framerate without oscillations, it is best to pick a fixed `deltaTime` value based on the average framerate your game is getting on iOS. Either 1:1 or half the frame rate is recommended. For example, if you get 30 fps, you should use 15 or 30 fps for fixed frame rate (0.033 or 0.066)

Can I access the gallery, music library or the native iPod player in Unity iOS?

A: Yes, if you implement it. Unity iPhone supports the **native plugin** system, where you can add any feature you need – including access to Gallery, Music library, iPod Player and any other feature that the iOS SDK exposes. Unity iOS does not provide an API for accessing the listed features through Unity **scripts**.

UnityGUI Considerations

What kind of performance impact will UnityGUI make on my games?

A: UnityGUI consumes more resources when more controls are used. It is ideal to limit your use of UnityGUI to game menus or very minimal GUI Controls while your game is running. It is important to note that every object with a script containing an `OnGUI()` call will require additional processor time – even if it is an empty `OnGUI()` block. It is best to disable any scripts that have an `OnGUI()` call if the GUI Controls are not being used. You can do this by marking the script as `enabled = false`.

Any other tips for using UnityGUI?

A: Try using GUILayout as little as possible. If you are not using GUILayout at all from one `OnGUI()` call, you can disable all GUILayout **rendering** using `MonoBehaviour.useGUILayout = false;` This doubles GUI rendering performance. Finally, use as few GUI elements while rendering 3D **scenes** as possible.

2018–06–14 Page amended with limited editorial review

iOS account setup

Leave feedback

A free Apple ID is all you need to test your build on any **iOS** devices you own. However, you should be aware that the free option is limited: you can't use services such as Game Center or In-App Purchases, and you can't submit your game to the App Store. You must register with the Apple Developer Program to publish your game.

If you don't yet have an Apple ID, get one from the Apple ID site, or register with the Apple Developer Program.

Install the most up-to-date version of Xcode (available from the Mac App Store). Xcode is a Mac-only integrated development environment (IDE) containing a suite of software development tools developed by Apple for developing software for OS X, iOS, WatchOS and tvOS.

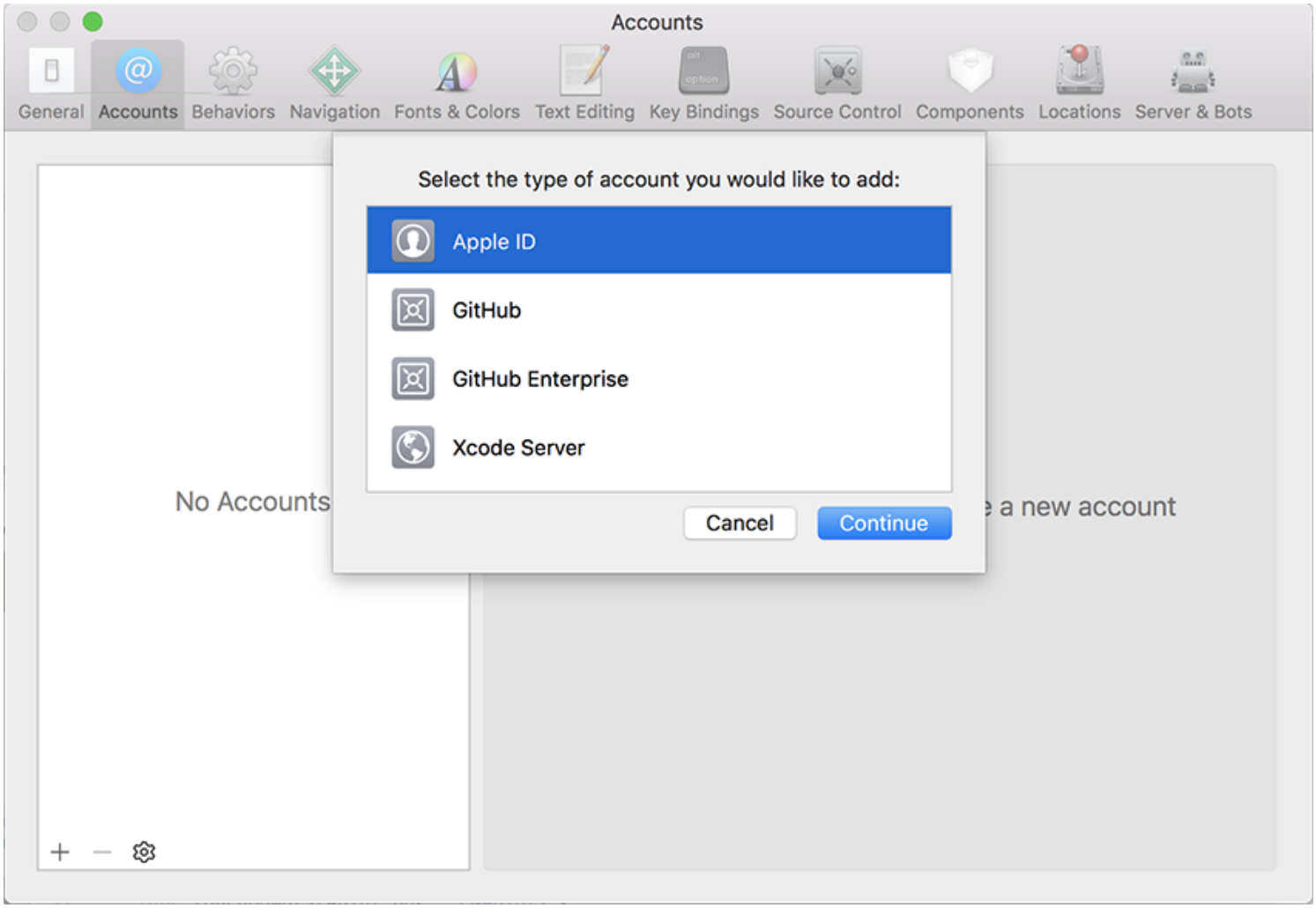
Adding your Apple ID to Xcode

Open Xcode.

From the menu bar select **Xcode** > **Preferences** to open the Preferences window.

Choose **Accounts** at the top of the window to display information about the Apple IDs that have been added to Xcode.

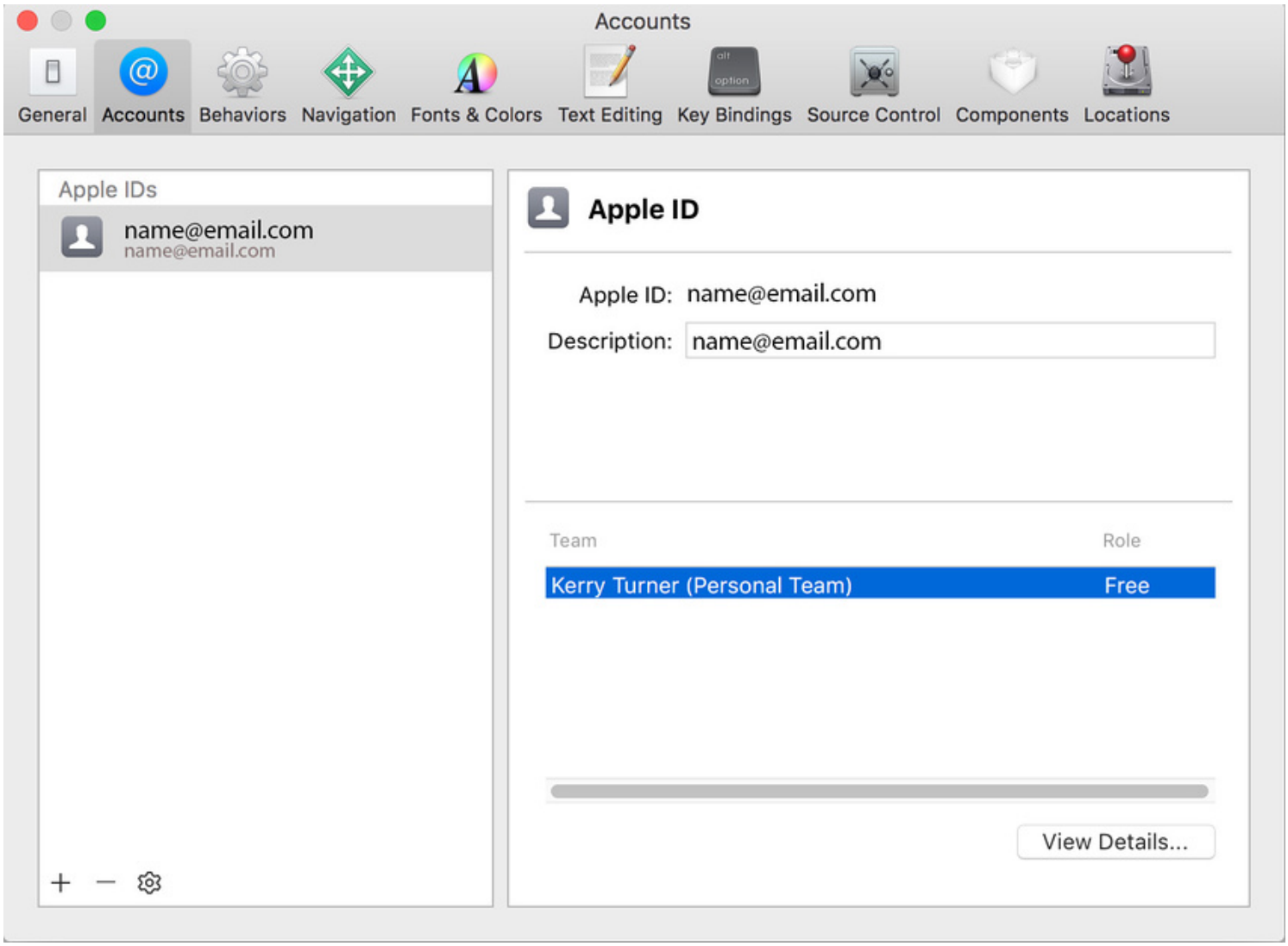
Click the plus sign in the bottom-left corner and choose **Apple ID** from the window that appears and click **Continue**.



Enter your Apple ID in the pop-up window that appears, and click **Next**.

Enter your password, and click **Next**.

Select your Apple ID in the list to see more information about it.



The **Team** section contains a list of all Apple Developer Program teams that you are a part of. If you're using a free Apple ID that isn't enrolled in the Apple Developer Program, a Personal Team is automatically created using your name.

Xcode can automatically configure your certificates, provisioning profiles and devices. If you're enrolled in the Apple Developer Program, you can use the Apple Developer Portal to do this manually.

2018–04–16 Page amended with limited editorial review

Inside the iOS build process

Leave feedback

Editor Based Building and Running

The iPhone/iPad application build process is a two step process:

XCode project is generated by Unity with all the required libraries, precompiled .NET code and serialized **assets**.
XCode project is built by XCode and deployed and run on the actual device.
When “Build” is hit on “Build settings” dialog only the first step is accomplished. Hitting “Build and Run” performs both steps. If in the project save dialog the user selects an already existing folder an alert is displayed. Currently there are two XCode project generation modes to select:

replace - all the files from target folder are removed and the new content is generated
append - the “Data”, “Libraries” and project root folder are cleaned and filled with newly generated content. The XCode project file is updated according to the latest Unity project changes. XCode project “Classes” subfolder could be considered as safe place to place custom native code, but making regular backups is recommended. Append mode is supported only for the existing XCode projects generated with the same Unity **iOS** version.
If Cmd+B is hit then the automatic build and run process is invoked and the latest used folder is assumed as the build target. In this case **append** mode is assumed as default.

Note: Step one above can be performed on a PC or a Mac. Only a Mac can perform Step two. This means to get a Unity project running on an iDevice you will need to have a Mac.

Command Line Building

Once Unity has been used to build the XCode project it is possible to perform the build and run from the command line. After the editor has built the XCode project do the following from the terminal:

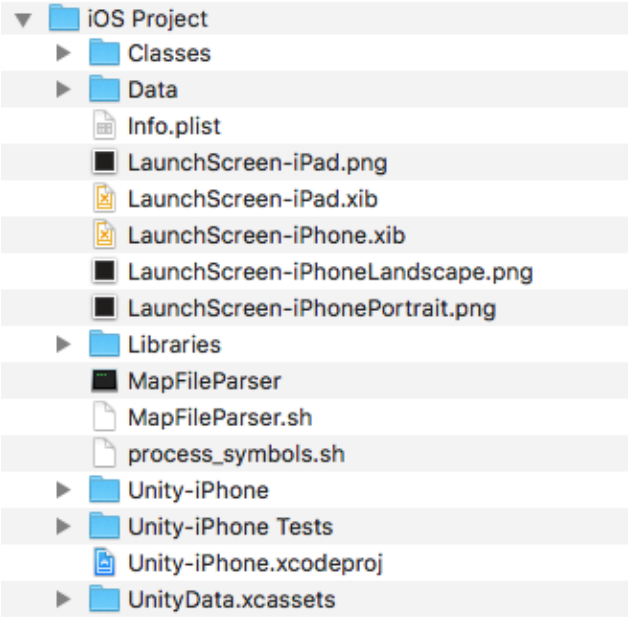
```
unity$ xcodebuild test -destination "platform=iOS,id=400d20d00baf8d4997b47be0416
```

Note that 400d20d00baf8d4997b47be0416cf5c44dd2d3bc in the command line example above is the ID of the iDevice that will run the project. You will need to determine your device ID using the **Window** > **Devices** menu in XCode.

Structure of a Unity XCode Project

Leave feedback

When you build a project for the **iOS** platform, Unity will create a folder that contains an XCode project. This project is required to compile and sign your app before deploying on devices, and it allows you to prepare and bundle your game for distribution on the App Store.



Before building the iOS project make sure that you set the **Bundle Identifier** in **Player Settings**. You may also choose the **SDK version** to run the game on the device or simulator.

Classes folder

This contains code that integrates the Unity Runtime and Objective-C. The contained files `main.mm` and `UnityAppController.mm/h` are the entry point of the application, and you can create your own AppDelegate, derived from UnityAppController. If you have **plugins** including `AppController.h` you can simply include `UnityAppController.h` instead. If you have `AppController.mm/h` in Plugins/iOS folders you can merge and rename them.

Also the `InternalProfiler.h` file defines a compiler conditional to enable the Internal **Profiler**. This is a folder for code that doesn't change often, and you can place your custom classes here. The changes to this folder are preserved between builds when the append mode is selected, but this function doesn't support multiple build targets and requires a fixed structure of the `Libraries` folder.

The Internal Profiler is fast and unobtrusive, and feeds basic information:

which subsystem is taking most of the frame time,
.NET heap size,
GC event count/duration. See built-in profiler for further information.

Data folder

This contains the serialized game **assets**, and .NET assemblies (`dll` or `dat` files) as full code (or metadata if stripping is on). The `machine.config` file is the setup for various .NET services such as security, WebRequest, and more. The content of this folder is refreshed with each build, and you should not modify it.

Libraries folder

This contains the .NET assemblies translated into ARM assembler (`s` files) or `libil2cpp.a` if the project was built using **IL2CPP**. The `libiPhone-lib.a` file is the Unity Runtime static library, and `RegisterMonoModules.cpp` binds Unity native code with .NET. The content of this folder is refreshed with each build, and you should not modify it.

Other newly created custom folders

Your custom files can be placed here.

Graphic files

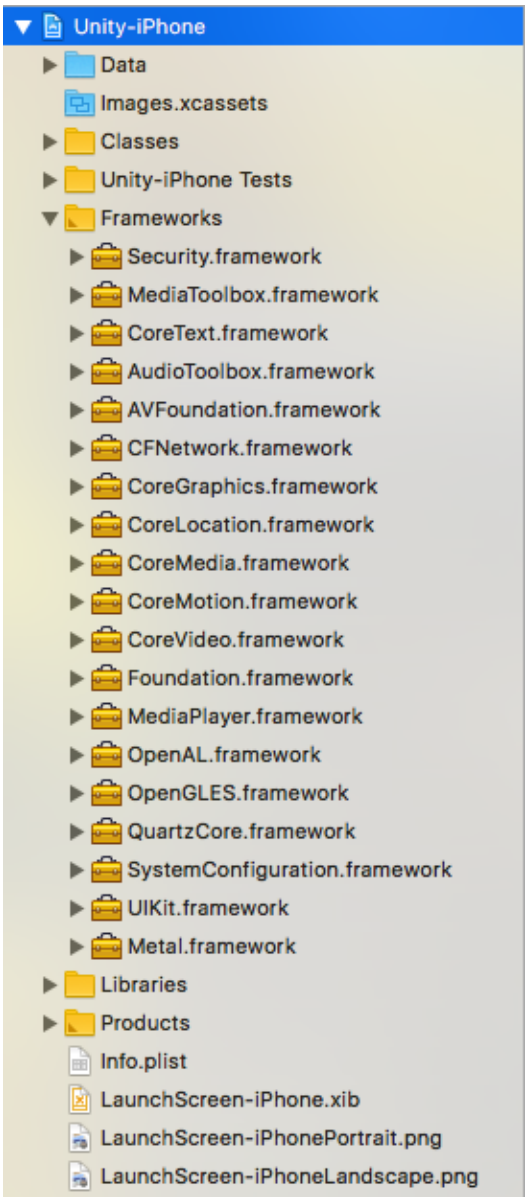
Icons and splash screens (`png` files) are kept in asset catalogs found in the `Unity-iPhone` folder. These files are automatically managed by Unity. Launch Screens, their XML Interface Builders (`xib` files) and Storyboard files are stored in the project's root folder. You can set them up in **Player Settings**. When creating custom launch images, make sure you adhere to Apple's Human Interface Guidelines.

Property List file

The `Info.plist` is managed via **Player Settings** in Unity. This file is updated when building the player, instead of being replaced. You should not modify it unless it is really needed.

Other files

These include the XCode Project file (`xcodeproj` file), and framework links that are only shown in the **Project Navigator**.



2018–06–14 Page amended with limited editorial review

Customizing an iOS Splash Screen

Leave feedback

If you use Unity Personal Edition to make your project, a default splash screen displays while your game loads. It is oriented according to the **Default Screen Orientation** option in the Player Settings. You cannot change this splash screen.

Unity Professional Edition users can change the splash screen. You can use any texture in the project as a splash screen.

The size of the texture depends on the target device:

320x480 **pixels** for 1–3rd gen devices
1024x768 for iPad mini/iPad 1st/2nd gen
2048x1536 for iPad 3th/4th gen
640x960 for 4th gen iPhone / iPod devices
640x1136 for 5th gen devices
750x1334 for 6th gen regular models
1242x2208 for iPhone Plus models
1125x2436 for iPhone X
The texture you supply is scaled to fit if necessary.

You can set the splash screen textures using the iOS Player Settings.

2018–06–14 Page amended with limited editorial review

iOS Hardware Guide

Leave feedback

Hardware models

The following list summarizes **iOS** hardware available in devices of various generations. Current device **shader** performance can compared on gfxbench which compares different hardware features using benchmarks.

iPhone Models

iPhone 4S

Screen: 960x640 **pixels**, LCD at 326 ppi, 800:1 contrast ratio.
Apple A5 Dual-Core 1 GHz ARM Cortex-A9 MPCore CPU
Dual-Core PowerVR SGX543MP2 GPU
512MB of memory
Rear 8.0 MP infra-red cut-off filter, back-illuminated sensor, 1080p HD videos at 30 **fps**.
Front 0.3 MP (VGA) with geotagging, tap to focus, and 480p SD video at 30 fps
GPS support
Compass Support

The iPhone 4S, with the A5 chip, is capable of **rendering** complex shaders throughout the entire screen. Even image effects may be possible. However, optimizing your shaders is still crucial. But if your game isn't trying to push limits of the device, optimizing scripting and gameplay is probably as much of a waste of time on this generation of devices as it is on PC.

iPhone 5

Screen: 1136x640 pixels, LCD at 326 ppi.
Apple A6 Dual-Core 1.3 GHz Apple-designed ARMv7s CPU
Triple-Core PowerVR SGX543MP3 GPU
1GB LPDDR2 of memory
Rear 8.0 MP infra-red cut-off filter, back-illuminated sensor, 1080p HD videos at 30 fps.
Front 1.2 MP tagging, tap to focus, and 720p SD video at 30 fps
GPS support
Compass Support

iPhone 5S

Screen: 4" 1136x640 pixels, LCD at 326 ppi.
Apple A7 Dual-Core 1.3 GHz Apple-designed ARMv8 64-bit CPU
M7 Motion Coprocessor
Four Cluster PowerVR G643 GPU
1GB LPDDR3 of memory
Rear 8.0 MP infra-red cut-off filter, back-illuminated sensor, 1080p HD videos at 30 fps.
Front 1.2 MP tagging, tap to focus, and 720p SD video at 30 fps
GPS and GLONASS support
Compass Support
Three-axis gyro
Proximity sensor
Ambient light sensor

Touch ID Fingerprint identity sensor

iPhone 6(+)

Screen iPhone6: 4.7" 1134x750 pixels, LCD at 326 ppi.
Screen iPhone6+: 5.5" 1920x1080 pixels, LCD at 401 ppi.
Apple A8 Dual-Core 1.4 GHz Apple-designed ARMv8-A 64-bit CPU
M8 motion coprocessor
Quad-Core PowerVR GX6450 GPU
1GB LPDDR3 of memory
Rear 8.0 MP infra-red cut-off filter, back-illuminated sensor, 1080p HD videos at 60 fps.
Front 1.2 MP tagging, tap to focus, and 720p SD video at 30 fps
GPS and GLONASS support
Compass Support
Three-axis gyro
Proximity sensor
Ambient light sensor
Touch ID Fingerprint identity sensor
NFC

iPhone 6s(+)

Screen iPhone6: 4.7" 1134x750 pixels, LCD at 326 ppi.
Screen iPhone6+: 5.5" 1920x1080 pixels, LCD at 401 ppi.
Apple A9 1.84 GHz Apple-designed ARMv8-A 64-bit CPU
Embedded M9 motion coprocessor
Hexa-Core PowerVR GT7600 GPU
2GB LPDDR4 of memory
Rear 12.0 MP phase detection autofocus, dual-LED dual-tone flash, 4K videos at 30 fps.
Front 5.0 MP face detection, **HDR**, panorama and 1080p HD video at 30 fps
A-GPS and GLONASS support
Compass Support
Three-axis gyro
Proximity sensor
Ambient light sensor
Touch ID Fingerprint identity sensor
NFC

iPhone 7(+)

Screen iPhone6: 4.7" 1134x750 pixels, LCD at 326 ppi.
Screen iPhone6+: 5.5" 1920x1080 pixels, LCD at 401 ppi.
Apple A10 Fusion 2.34 GHz Quad-core (2 Hurricane + 2 Zephyr)
M10 motion coprocessor
Hexa-Core PowerVR Series7XT Plus
iPhone7: 2GB LPDDR4 of memory
iPhone7+: 3GB LPDDR4 of memory
Rear 12.0 MP phase detection autofocus, dual-LED dual-tone flash, 4K videos at 30 fps.
iPhone7+ also has a second 12.0 MP **camera**.
Front 7.0 MP face detection, HDR, panorama and 1080p HD video at 30 fps
A-GPS and GLONASS support
Compass Support
Three-axis gyro
Proximity sensor
Ambient light sensor
Touch ID Fingerprint identity sensor
NFC

iPhone 8(+)

Screen iPhone6: 4.7" 1134x750 pixels, LCD at 326 ppi.
Screen iPhone6+: 5.5" 1920x1080 pixels, LCD at 401 ppi.
Apple A11 Bionic 2.39 GHz Hexa-core (2 Monsoon + 4 Mistral) with Neural Engine
M11 motion coprocessor
iPhone8: 2GB LPDDR4 of memory
iPhone8+: 3GB LPDDR4 of memory
Rear 12.0 MP phase detection autofocus, dual-LED dual-tone flash, 4K videos at 60 fps.
iPhone8+ also has a second 12.0 MP camera.
Front 7.0 MP face detection, HDR, panorama and 1080p HD video at 30 fps
A-GPS and GLONASS support
Compass Support
Three-axis gyro
Proximity sensor
Ambient light sensor
Touch ID Fingerprint identity sensor
NFC
Wireless charging supported

iPhone X

Screen: 5.8" 1125x2436 pixels, OLED at 458 ppi.
Apple A11 Bionic 2.39 GHz Hexa-core (2 Monsoon + 4 Mistral) with Neural Engine
M11 motion coprocessor
GB LPDDR4 of memory
Rear Dual 12.0 MP phase detection autofocus, dual-LED dual-tone flash, 4K videos at 60 fps.
Front 7.0 MP face detection, HDR, panorama and 1080p HD video at 30 fps
A-GPS and GLONASS support
Compass Support
Three-axis gyro
Proximity sensor
Ambient light sensor
Touch ID Fingerprint identity sensor
NFC
Wireless charging supported

iPod Touch Models

iPod Touch 5th generation

Screen: 1136x640 pixels, LCD at 326 ppi
Apple A5 Dual-Core 1GHz (underclocked to 800MHz) ARM Cortex-A9 MPCore CPU
Dual-Core PowerVR SGX543MP2 GPU
512MB of memory
Rear 5.0 MP backside illuminated CMOS image sensor with 1080p HD video at 30 fps, face detection and video stabilization.
Front 1.2 MP with geotagging, tap to focus, and 720p HD video at 30 fps

iPad Models

iPad 2

Screen: 1024x768 pixels, LCD at 132 ppi, LED-backlit.
Apple A5 Dual-Core 1 GHz ARM Cortex-A9 MPCore CPU
Dual-Core PowerVR SGX543MP2 GPU
512MB DDR2 Ram
GPS support
Accelerometer, Three-axis Gyro, Proximity Sensor, Ambient Light Sensor, Magnetometer
Wifi + Bluetooth 2.1 + (3G Cellular HSDPA, 2G cellular EDGE on the 3G version)
Mechanical keys: Home, sleep, function switch, volume.

iPad2: the A5 can do full screen bumpmapping, assuming the shader is simple enough. However, it is likely that your game will perform best with bumpmapping only on crucial objects. Full screen image effects still out of reach. Scripting optimization less important.

iPad (3rd generation)

Screen: 2048x1536 pixels, LCD at 264 ppi, LED-backlit.
Apple A5X
Dual-Core 1 GHz ARM Cortex-A9 MPCore CPU
Quad-Core PowerVR SGX543MP4 GPU
1GB LPDDR2 Ram
GPS and GLONASS support
Accelerometer, Three-axis Gyro, Proximity Sensor, Ambient Light Sensor, Magnetometer
Wifi + Bluetooth 4.0 + (LTE, 3G Cellular HSDPA, 2G cellular EDGE on the 3G version)
Mechanical keys: Home, sleep, function switch, volume.

The iPad 3 has been shown to be capable of render-to-texture effects such as reflective water and fullscreen image effects. However, optimized shaders are still crucial. But if your game isn't trying to push limits of the device, optimizing scripting and gameplay is probably as much of a waste of time on this generation of devices as it is on PC.

iPad (4th generation)

Screen: 2048x1536 pixels, LCD at 264 ppi, LED-backlit.
Apple A6X Dual-Core 1.4 GHz Apple Swift
Quad-Core PowerVR SGX554MP4 GPU
1GB LPDDR2 Ram
GPS and GLONASS support
Accelerometer, Three-axis Gyro, Proximity Sensor, Ambient Light Sensor, Magnetometer
Wifi + Bluetooth 4.0 + (LTE, 3G Cellular HSDPA, 2G cellular EDGE on the 3G version)
Mechanical keys: Home, sleep, function switch, volume.

iPad (5th generation)

Screen: 2048x1536 pixels, LCD at 264 ppi, LED-backlit.

Apple A9 1.85 GHz dual-core 64-bit ARMv8-A
PowerVR 7XT GT7600
M9 Motion Coprocessor
2GB LPDDR4 Ram
A-GPS and GLONASS support
Accelerometer, Three-axis Gyro, Proximity Sensor, Ambient Light Sensor, Magnetometer
Wifi + Bluetooth 4.2 + (LTE Advanced, UMTS/HSPA/HSPA+/DC-HSDPA, GSM/EDGE)
Mechanical keys: Home, sleep, function switch, volume.

iPad (6th generation)

Screen: 2048x1536 pixels, LCD at 264 ppi, LED-backlit.
Apple A10 Fusion 2.34 GHz Quad-core
PowerVR 7XT GT7600
M10 Motion Coprocessor
2GB LPDDR4 Ram
A-GPS and GLONASS support
Apple Pencil support
Accelerometer, Three-axis Gyro, Proximity Sensor, Ambient Light Sensor, Magnetometer
Wifi + Bluetooth 4.2 + (LTE Advanced, UMTS/HSPA/HSPA+/DC-HSDPA, GSM/EDGE)
Mechanical keys: Home, sleep, function switch, volume.

iPad Air

Screen: 2048x1536 pixels, LCD at 264 ppi, LED-backlit.
Apple A7 Dual-Core 1.4 GHz Apple Cyclone
Quad-Core PowerVR G6430 GPU
M7 Motion Coprocessor
1GB LPDDR3 Ram
GPS and GLONASS support
Accelerometer, Three-axis Gyro, Proximity Sensor, Ambient Light Sensor, Magnetometer
Wifi + Bluetooth 4.0 + (LTE, 3G Cellular HSDPA, 2G cellular EDGE on the 3G version)
Mechanical keys: Home, sleep, function switch, volume.

iPad Air 2

Screen: 2048x1536 pixels, LCD at 264 ppi, LED-backlit.
Apple A8X 1.5 GHz triple-core
Hex-Core PowerVR GX6650 GPU
M8 Motion Coprocessor
2GB LPDDR3 Ram
GPS and GLONASS support
Accelerometer, Three-axis Gyro, Proximity Sensor, Ambient Light Sensor, Magnetometer
Wifi + Bluetooth 4.0 + (LTE, 3G Cellular HSDPA, 2G cellular EDGE on the 3G version)
Mechanical keys: Home, sleep, function switch, volume.

iPad Mini

Screen: 1024x768 pixels, LCD at 163 ppi, LED-backlit.
Apple A5 Dual-Core 1 GHz ARM Cortex-A9
Dual-Core PowerVR SGX543MP2 GPU
512MB DDR2 Ram
GPS and GLONASS support
Accelerometer, Three-axis Gyro, Proximity Sensor, Ambient Light Sensor, Magnetometer
Wifi + Bluetooth 4.0 + (LTE, 3G Cellular HSDPA, 2G cellular EDGE on the 3G version)
Mechanical keys: Home, sleep, function switch, volume.

iPad Mini 2

Screen: 2048x1536 pixels, LCD at 326 ppi, LED-backlit.
Apple A7 Dual-Core 1.3 GHz Apple Cyclone

Quad-Core PowerVR G6430 GPU
1GB LPDDR3 Ram
GPS and GLONASS support
Accelerometer, Three-axis Gyro, Proximity Sensor, Ambient Light Sensor, Magnetometer
Wifi + Bluetooth 4.0 + (LTE, 3G Cellular HSDPA, 2G cellular EDGE on the 3G version)
Mechanical keys: Home, sleep, function switch, volume.

iPad Mini 3

Screen: 2048x1536 pixels, LCD at 326 ppi, LED-backlit.
Apple A7 Dual-Core 1.3 GHz Apple Cyclone
Quad-Core PowerVR G6430 GPU
1GB LPDDR3 Ram
GPS and GLONASS support
Accelerometer, Three-axis Gyro, Proximity Sensor, Ambient Light Sensor, Magnetometer
Wifi + Bluetooth 4.0 + (LTE, 3G Cellular HSDPA, 2G cellular EDGE on the 3G version)
Mechanical keys: Home, sleep, function switch, volume.

iPad Pro (1st Generation)

Screen 9.7”: 20148x1536 pixels, LCD at 264 ppi, LED-backlit.
Screen 12.9”: 2732x2048 pixels, LCD at 264 ppi, LED-backlit.
Apple A9X 64-bit ARMv8-A dual-core (9.7” model is slightly underclocked)
PowerVR 7XT GT7800 (12 Clusters)
M9 Motion Coprocessor
12.9” Model: 4GB LPDDR4 Ram
9.7” Model: 2GB LPDDR4 Ram
A-GPS and GLONASS support
Accelerometer, Three-axis Gyro, Proximity Sensor, Ambient Light Sensor, Magnetometer
Wifi + Bluetooth 4.2 + (LTE Advanced, UMTS/HSPA/HSPA+/DC-HSDPA, GSM/EDGE)
Mechanical keys: Home, sleep, function switch, volume.

iPad Pro (2nd Generation)

Screen 10.5”: 2224x1668 pixels, LCD at 264 ppi, LED-backlit.
Screen 12.9”: 2732x2048 pixels, LCD at 264 ppi, LED-backlit.
Apple A10X Fusion 2.38 GHz Hexa-core (3 Hurricane + 3 Zephyr)
PowerVR 7XT GT7600 Plus (12 Clusters)
M10 Motion Coprocessor
4GB LPDDR4 Ram
A-GPS and GLONASS support
ProMotion support
Accelerometer, Three-axis Gyro, Proximity Sensor, Ambient Light Sensor, Magnetometer
Wifi + Bluetooth 4.2 + (LTE Advanced, UMTS/HSPA/HSPA+/DC-HSDPA, GSM/EDGE)
Mechanical keys: Home, sleep, function switch, volume.

Graphics Processing Unit and Hidden Surface Removal

The iPhone/iPad graphics processing unit (GPU) is a Tile-Based Deferred Renderer. In contrast with most GPUs in desktop computers, the iPhone/iPad GPU focuses on minimizing the work required to render an image as early as possible in the processing of a **scene**. That way, only the visible pixels will consume processing resources.

The GPU’s frame buffer is divided up into tiles and rendering happens tile by tile. First, triangles for the whole frame are gathered and assigned to the tiles. Then, visible fragments of each triangle are chosen. Finally, the selected triangle fragments are passed to the rasterizer (triangle fragments occluded from the camera are rejected at this stage).

In other words, the iPhone/iPad GPU implements a **Hidden Surface Removal** operation at reduced cost. Such an architecture consumes less memory bandwidth, has lower power consumption and utilizes the texture cache better. Tile-Based Deferred Rendering allows the device to reject occluded fragments before actual rasterization, which helps to keep overdraw low.

For more information, see also:

Apple Notes on iPhone/iPad GPU and OpenGL ES
Apple Performance Advices for OpenGL ES in General
Apple Performance Advices for OpenGL ES Shaders

Texture compression

iOS devices support these texture **compression** formats:

PVRTC provides support for RGB and RGBA (color information plus an alpha channel) **texture formats** and can compress a single pixel to two or four bits. The PVRTC format is essential to reduce the memory footprint and to reduce consumption of memory bandwidth, as in the rate at which data can be read from memory, which is usually very limited on mobile devices.

ETC compresses 4x4 pixel blocks into 64-bit quantities, but lacks an alpha channel.

ETC2 supports both 1-bit and 8-bit alpha channel compression.

ASTC is flexible format that uses pixel block sizes. It's supported by Metal and OpenGL ES 3.0 graphic APIs.

Note: the DXT **texture compression** format on iOS devices is NOT supported.

For detailed texture compression options in iOS, see iOS 2D Texture Overrides.

Vertex Processing Unit

The iPhone/iPad has a dedicated unit responsible for vertex processing which runs calculations in parallel with rasterization. In order to achieve better parallelization, the iPhone/iPad processes vertices one frame ahead of the rasterizer.

Unified Memory Architecture

Both the CPU and GPU on the iPhone/iPad share the same memory. The advantage is that you don't need to worry about running out of video memory for your textures (unless, of course, you run out of main memory too). The disadvantage is that you share the same memory bandwidth for gameplay and graphics. The more memory bandwidth you dedicate to graphics, the less you will have for gameplay and physics.

Multimedia CoProcessing Unit

The iPhone/iPad main CPU is equipped with a powerful SIMD (Single Instruction, Multiple Data) coprocessor supporting either the **VFP** or the **NEON** architecture. The Unity iOS run-time takes advantage of these units for multiple tasks such as calculating skinned **mesh** transformations, geometry batching, audio processing and other calculation-intensive operations.

2018–06–28 Page amended with limited editorial review

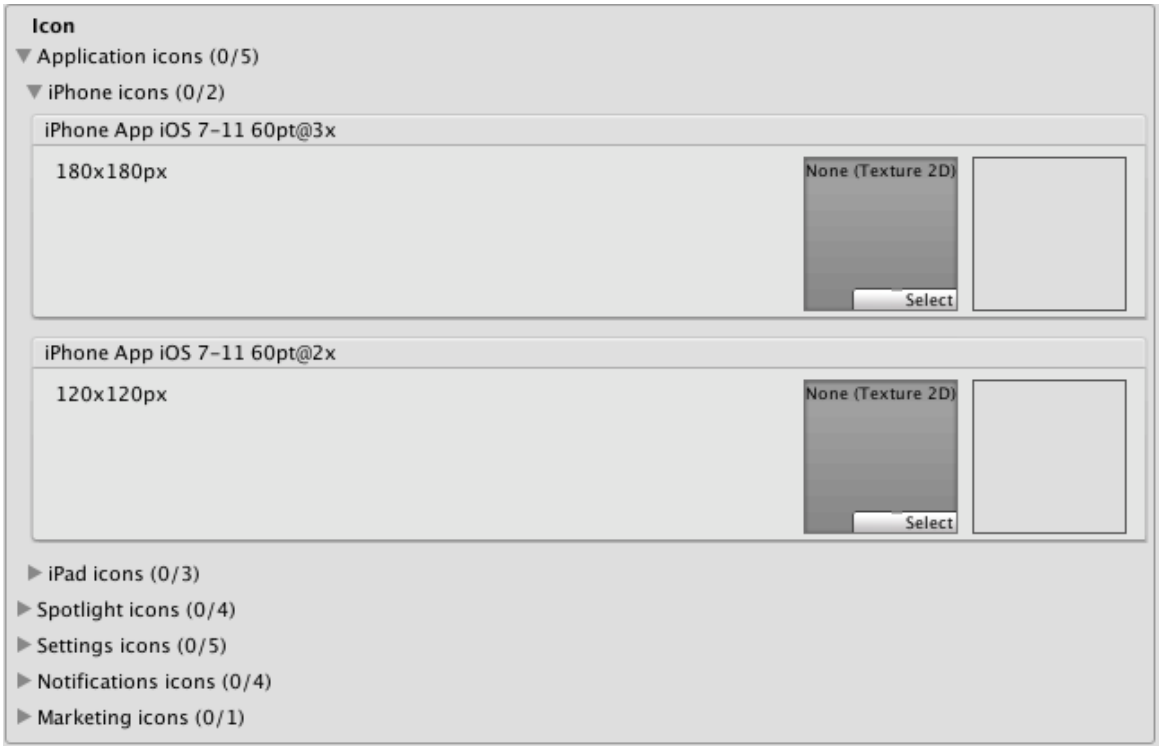
iOS Player Settings

Leave feedback

This page details the **Player Settings** specific to **iOS**. A description of the general Player Settings can be found here.

Note that Unity iOS requires iOS 7.0 or higher. Unity does not support iOS 6.0 or earlier versions.

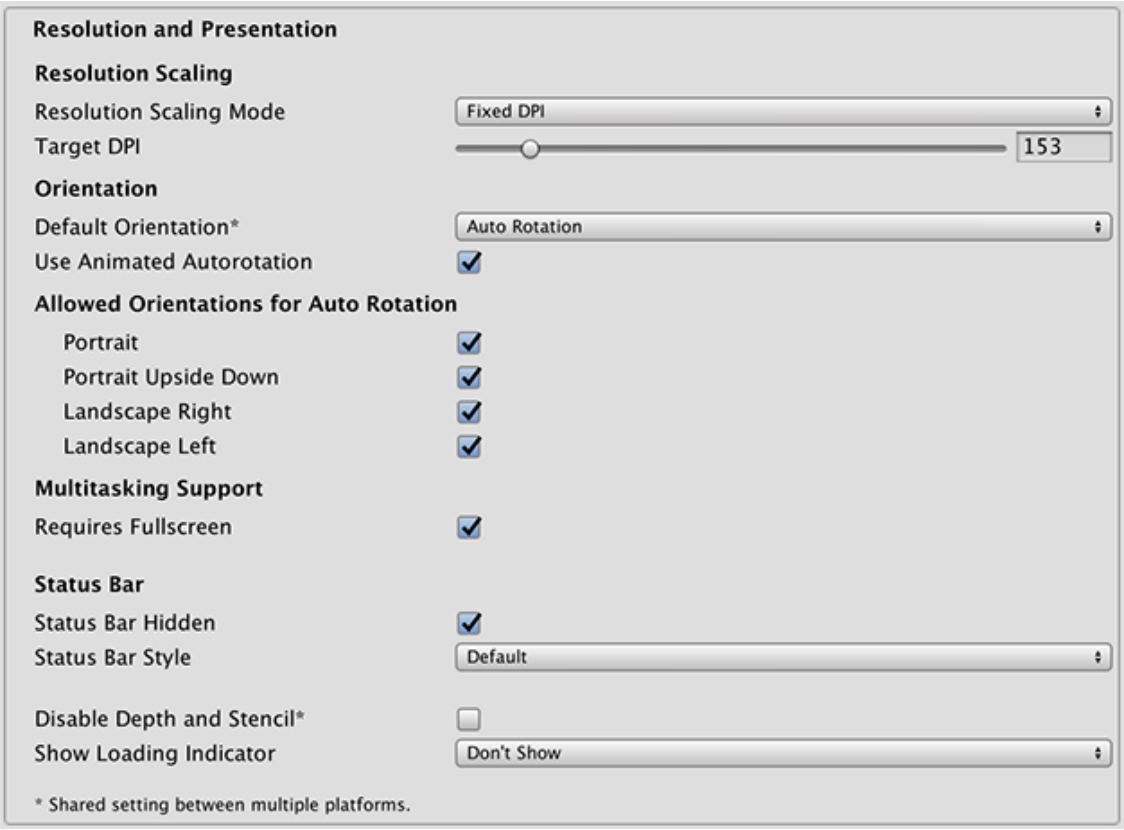
Icon



Note: If any icon textures are omitted, the icon texture with the nearest size is scaled accordingly (preference is given to larger resolution textures).

Property:	*Function:*
Application icons	Assign the custom icon that you would like to appear on your app for each iOS device.
Spotlight icons	Assign the custom icon that you would like to appear in the Spotlight search results for your game for each iOS device.
Settings icons	Assign the custom icon that you would like to appear on the phone's main Settings page for each iOS device.
Notification icons	Assign the custom icon that you would like to appear when sending notifications from your game for each iOS device.
Marketing icons	Assign the custom icon that you would like to appear for your game in the App Store for each iOS device.

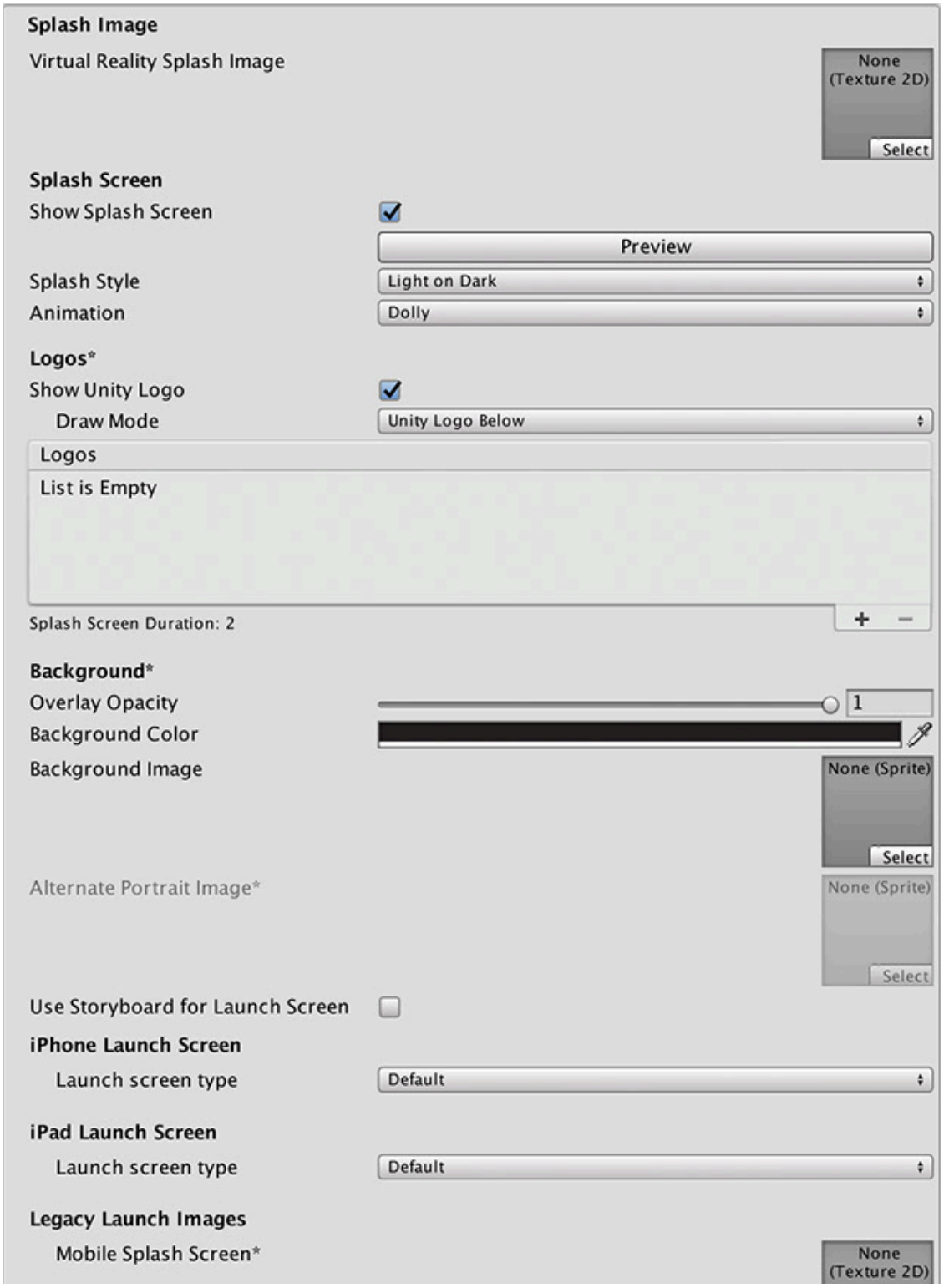
Resolution and Presentation

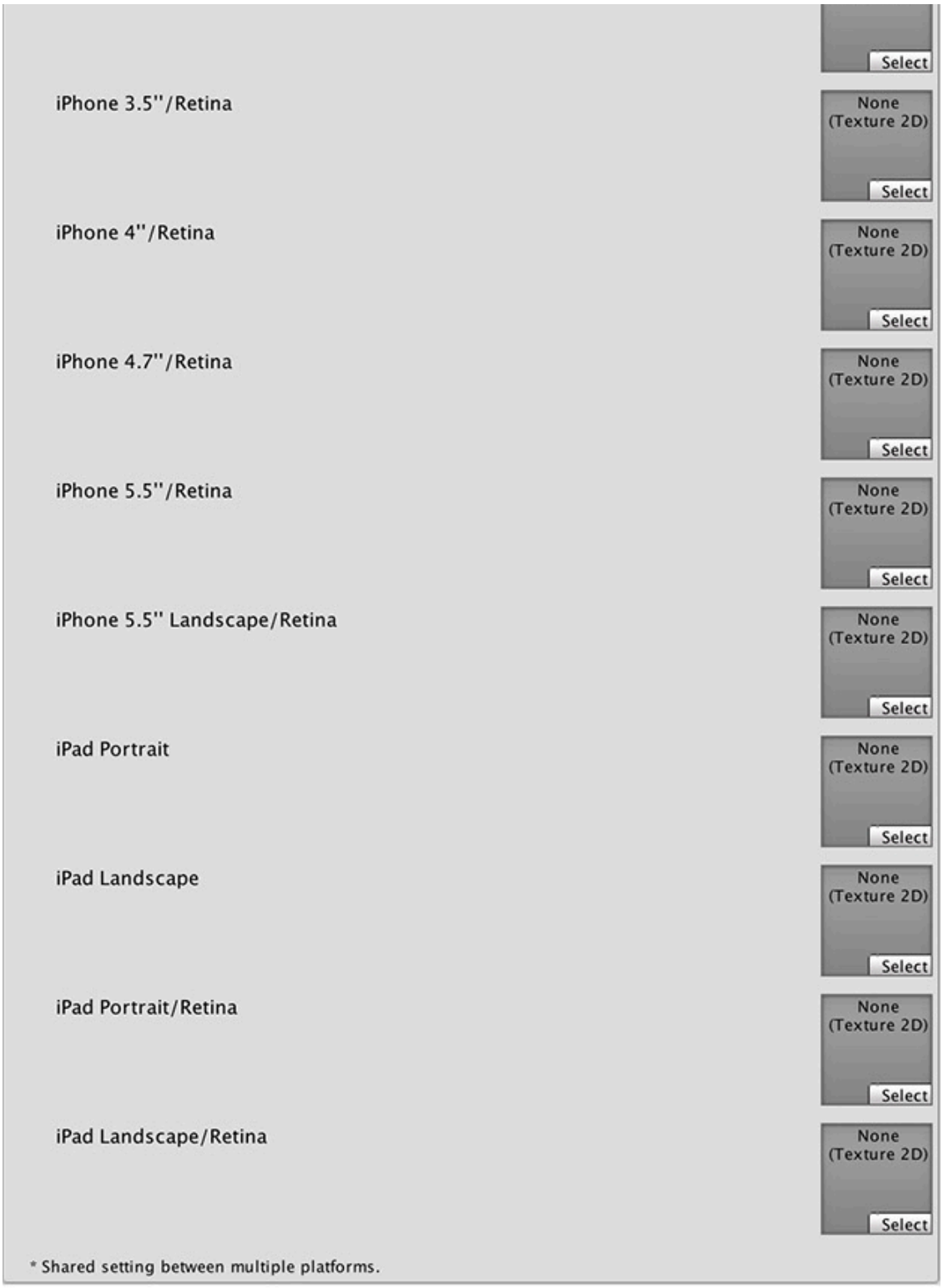


Property:	*Function:*
Resolution Scaling	
Resolution Scaling Mode	Set the scaling to be equal to or below the native screen resolution: **FixedDPI** allows you to scale the device's screen resolution below its native resolution and show the **Target DPI** property. Use this to optimize performance and battery life or target a specific DPI setting. **Disabled** ensures that scaling is not applied and the game renders to its native screen resolution.
Target DPI	Set the target DPI of the game screen. The game screen is downscaled to match this setting if the device's native screen DPI is higher than this value. This option only appears when the **Resolution Scaling Mode** is set to **Fixed DPI**.
Orientation	
Default Orientation This setting is shared between iOS and Android devices	The game's screen orientation. The options are: **Portrait** (home button at the bottom), **Portrait Upside Down** (home button at the top), **Landscape Left** (home button on the right side), **Landscape Right** (home button on the left side), and **Auto Rotation** (screen orientation changes with device orientation)
Use Animated Autorotation	Check this box if you want orientation changes to animate the screen rotation rather than just switch. This is only visible when **Default Orientation** is set to **Auto Rotation**.
Allowed Orientations for Auto Rotation (only visible when **Default Orientation** is set to **Auto Rotation**.)	
Portrait	Allow portrait orientation.
Portrait Upside Down	Allow portrait upside-down orientation.
Landscape Right	Allow landscape right orientation (home button on the **left** side).
Landscape Left	Allow landscape left orientation (home button is on the **right** side).
Multitasking Support	
Requires Fullscreen	Check this box if your game requires fullscreen.

Property:	*Function:*
Status Bar	
Status Bar Hidden	Check this box to hide the the status bar when the application launches.
Status Bar Style	Define the style of the status bar when the application launches. The options are **Default**, **Black Translucent** and **Black Opaque**.
Disable Depth and Stencil	Check this box to disable the depth and stencil buffers.
Show Loading Indicator	Select how the loading indicator be displayed. The options are **Don't Show**, **White Large**, **White**, and **Gray**.

Splash Image





There are two ways to implement splash images on iOS: **Launch Images** and **Launch Screens**.

Launch Images

Launch Images are static splash screen images that occupy the entire screen.

Launch Images are defined in an **Asset** catalog (Images.xcassets/LaunchImage). Always add a **Launch Screen** for each supported size and orientation combination.

Only iPhone 6+ supports landscape orientation; other iPhones can only use portrait. Launch Images are selected in the following order:

The specific **Launch Image** override, if the texture is set
Default Unity splash screen launch image, which is a solid blue-black color
You need to set all **Launch Images** for your build.

Launch Screens

A **Launch Screen** is an XIB file from which iOS creates a splash screen dynamically on the device.

Launch Screens have a limitation: it is not possible to display different content depending on iPad device orientation. All iPhones support landscape **Launch Screens**; however, due to a bug in iOS, **Landscape Left** is shown instead of **Landscape Right** on certain iOS versions.

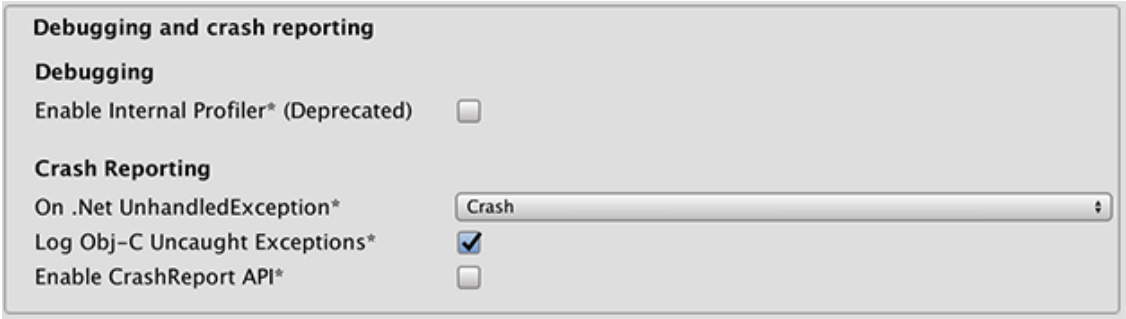
Note: For advice on cross-platform Splash Image properties, see Splash Screen settings.

Property:	*Function:*
Virtual Reality Splash Image	Specify the texture that should be used for the iOS splash screen on a **Virtual Reality** application.
Use Storyboard for Launch	Enable this option to show the **Custom Storyboard** button. Click the button to select a storyboard to show when your game starts up on the device. For your storyboard to appear here, you firstly need to create the storyboard in Xcode and copy it to your Project.
Legacy Launch Images	
Mobile Splash Screen	Specifies texture which should be used for iOS Splash Screen. Standard Splash Screen size is 320x480.(This is shared between Android and iOS.)
iPhone 3.5"/Retina	Specifies texture which should be used for iOS 3.5" Retina Splash Screen. Splash Screen size is 640x960.
iPhone 4"/Retina	Specifies texture which should be used for iOS 4" Retina Splash Screen. Splash Screen size is 640x1136.
iPhone 4.7"/Retina	Specifies texture which should be used for iOS 4.7" Retina Splash Screen. Splash Screen size is 750x1334.
iPhone 5.5"/Retina	Specifies texture which should be used for iOS 5.5" Retina Splash Screen. Splash Screen size is 1242x2208.
iPhone 5.5" Landscape/Retina	Specifies texture which should be used for iOS 5.5" Landscape/Retina Splash Screen. Splash Screen size is 2208x1242.
iPhone X\Retina	Specifies texture which should be used for iPhone X Retina Splash Screen. Splash Screen size is 1125x2436.
iPhone X Landscape\Retina	Specifies texture which should be used for iPhone X Landscape/Retina Splash Screen. Splash Screen size is 2436x1125.
iPad Portrait	Specifies texture which should be used as iPad Portrait orientation Splash Screen. Standard Splash Screen size is 768x1024.
iPad Landscape	Specifies texture which should be used as iPad Landscape orientation Splash Screen. Standard Splash Screen size is 1024x768.
iPad Portrait/Retina	Specifies texture which should be used as the iPad Retina Portrait orientation Splash Screen. Standard Splash Screen size is 1536x2048.
iPad Landscape/Retina	Specifies texture which should be used as the iPad Retina Landscape orientation Splash Screen. Standard Splash Screen size is 2048x1536.
Launch Screen type	Allows you to select between the **launch screen** types
- None	The behavior is as if only launch images are used.
- Default	A launch screen that is very much like a launch image. One image is selected for portrait and landscape. The selection order: iPhone 6+ launch images, shared mobile launch image, default Unity launch image for iPhone 6+. The images are displayed using aspect-fill mode.

Property:	*Function:*
- Image with background, relative size	A center-aligned image is shown, with the rest of area filled with solid color. The image size is user-specified percentage of the screen size, computed in the smaller dimension (vertical on landscape, horizontal in portrait orientations). User also specifies background color and images for portrait and landscape orientations. Image selection order: the user-specified image, shared mobile launch image, default Unity launch image for iPhone 6+. The images are displayed using aspect-fill mode.
- Image with background, constant size	Same as relative size option except that the size of the image is defined by user-specified number of points.
- Custom Xib	An user-specified XIB file from any location.

In Unity Personal Edition the Unity Splash Screen displays as soon as engine initializes, in addition to your chosen splash screen.

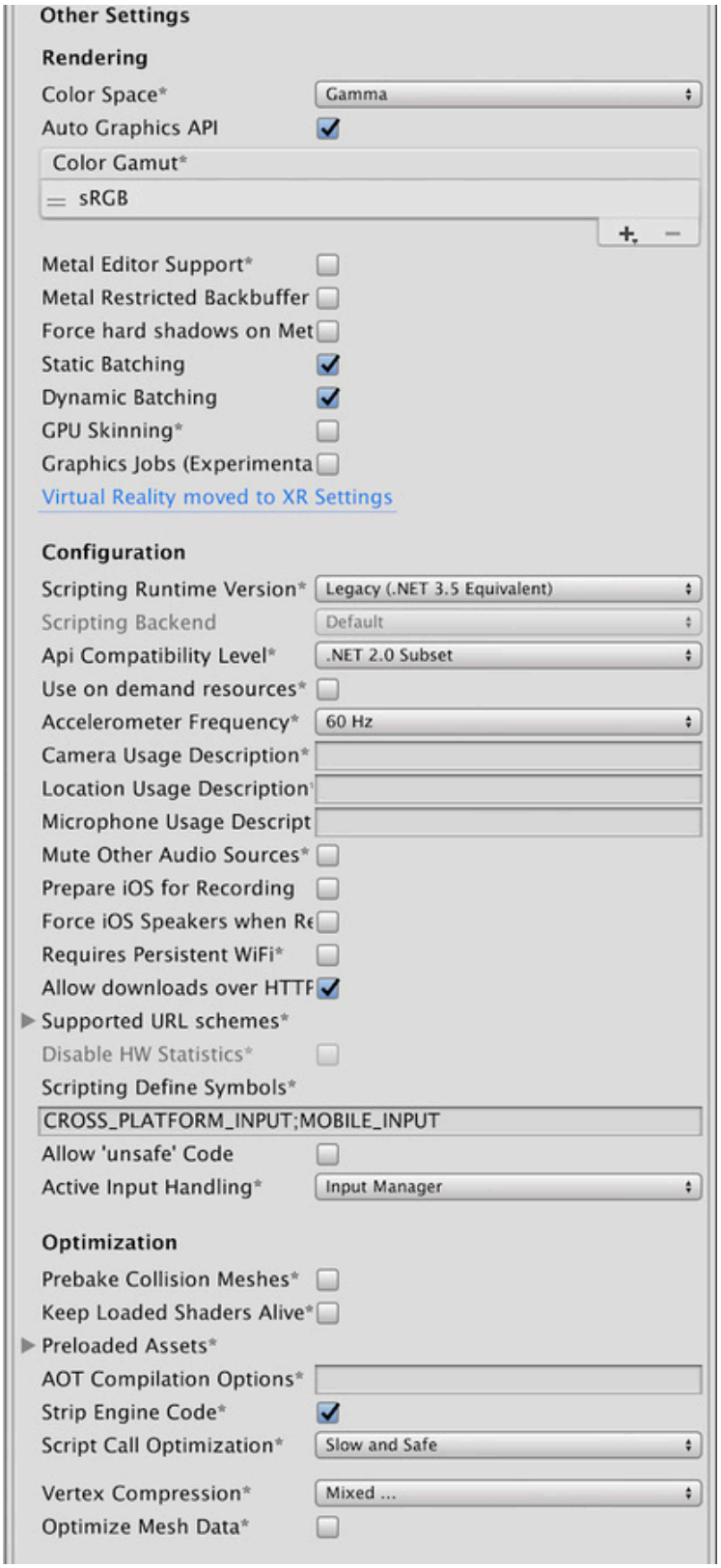
Debugging and crash reporting



Property:	*Function:*
Enable Internal Profiler__ (Deprecated)**|Enables an internal profiler which collects performance data of the application and prints a report to the console. The report contains the number of milliseconds that it took for each Unity subsystem to execute on each frame. The data is averaged across 30 frames.| |**On .Net UnhandledException**|The action taken on .NET unhandled exception. The options are** Crash__ (the application crashes hardly and forces iOS to generate a crash report that can be submitted to iTunes by app users and inspected by developers), **Silent Exit** (the application exits gracefully).	
Log ObjC uncaught exceptions	Enables a custom Objective-C Uncaught Exception handler, which will print exception information to console.

Property:	*Function:*
Enable Crash Report API	Enables a custom crash reporter to capture crashes. Crash logs will be available to **scripts** via CrashReport API.

Other Settings



Property:	*Function:*
Rendering	
Color Space	Make Unity use **Gamma** color space or **Linear** color space for rendering. For more information, see linear rendering documentation. **Linear** color space is only supported on Metal graphics API.

Property:	*Function:*
Auto Graphics API	Allows you to select which graphics API is used. When checked, Unity will include Metal, and GLES2 as a fallback for devices where Metal is not supported. When unchecked, you can manually pick and reorder the graphics APIs. Manually picking just one API will adjust your app's info.plist which will result in appropriate app store restrictions.
Graphics APIs	Set the graphics APIs to use for rendering. Click the + icon to add an API and the - icon to remove an API. When a device supports more than one Graphics API, Unity uses the first item in the ordered list. Click and drag an API in the list to change the order. This will only appear when **Auto Graphics API** is unchecked.
Color Gamut	Set the color gamut to use for rendering. When targeting recent iOS devices with wide color gamut displays, use **DisplayP3** to utilize full display capabilities. Use **Metal Editor Support** as a fallback for older devices.
Metal Editor Support	Makes the Unity Editor use the Metal API and unlocks faster **shader** iteration for targeting the Metal API. **Metal API Validation** can help catch graphics-related errors early, although it might decrease performance. For more information, see Validating Metal API.
Force hard shadows on Metal	Forces Unity to use point sampling for shadows on Metal. This reduces shadow quality, which should give better performance.
Metal Restricted Backbuffer Use	Allow improved performance in non-default device orientation. This sets the frameBufferOnly flag on the back buffer, which prevents readback from the back buffer but enables some driver optimization.
Multithreaded Rendering	Enable multithreaded rendering. This is only supported on Metal.
Static Batching	Set this to use Static batching on your build (enabled by default).
Dynamic Batching	Set this to use Dynamic Batching on your build.
GPU Skinning	Should DX11/ES3 GPU **skinning** be enabled?
Identification	
Bundle Identifier	The string used in your provisioning certificate from your Apple Developer Network account. (This is shared between iOS and Android.)
Bundle Version	Specifies the build version number of the bundle, which identifies an iteration (released or unreleased) of the bundle. The version is specified in the common format of a string containing numbers separated by dots (eg, 4.3.2).
Build	The build number can be entered here to allow you to keep track of the number of builds that have been made.
Signing Team ID	Set this property with your Apple Developer Team ID. You can find this on the Apple Developer website under Account > Membership. This sets the Team ID for the generated Xcode project, allowing developers to use the Build and Run functionality. An Apple Developer Team ID must be set here for automatic signing of your app. For more information, see Creating Your Team Provisioning Profile.
Automatically Sign	Enable this to allow Xcode to automatically sign your build.

Property:	*Function:*
iOS Provisioning Profile	Specify an Apple provisioning profile to allow your game to run on all your iOS devices and use app services. Click the **Browse** button to select a downloaded provisioning profile file. The **Profile ID** is the internal ID of the provisioning profile set in the Xcode project, for example: `1234abcd-12ab-12ab-12ab-1234abcd1234`. This appears automatically when you select a provisioning profile file, or you can enter it manually. Note: This property only appears when **Automatically Sign** is unchecked.
Configuration	
Scripting Runtime Version	Choose which .NET runtime to use in your project. For more details, see Microsoft's .NET documentation.
.NET 3.5 Equivalent	A .NET runtime which implements the .NET 3.5 API. This functionality is deprecated, and should no longer be used. Please use .NET 4.
.NET 4.x Equivalent	A .NET runtime which implements the .NET 4 API. This API is newer than .NET 3.5, and as such, it offers access to more APIs, is compatible with more external libraries, and supports C# 6. This is the default scripting runtime
Scripting Backend	Allows you to select between **IL2CPP** and Mono scripting backends. The default is **IL2CPP**, and in most normal situations there should be no reason to switch to the older Mono backend. Unless you are running into bugs specifically relating to **IL2CPP**, you should not select Mono. **Mono builds are no longer accepted in the App store and Mono is not supported by iOS 11 and above.**
Api Compatibility Level	Specifies active .NET API profile. See below:
- .Net 2.0	.Net 2.0 libraries. Maximum .net compatibility, biggest file sizes.
- .Net 2.0 Subset	Subset of full .net compatibility, smaller file sizes.
C++ Compiler Configuration	Choose whether to compile your IL2CPP generated code in Debug or Release mode:
- Debug	Use **Debug** mode for debugging because it turns off all optimizations. This makes the code quicker to build but slower to run.
- Release	Use **Release** mode for testing and delivery to users because it enables optimizations. This makes the compiled code run faster and the binary size smaller, but it takes longer to compile.
Use on Demand Resource	When enabled, allows you to use on-demand resources.
Accelerometer Frequency	How often is the accelerometer sampled? The options are **Disabled** (i.e. no samples are taken), **15Hz**, **30Hz**, **60Hz** and **100Hz**.
Camera Usage Description	Allows you to enter the reason for accessing the **camera** on the iOS device.
Location Usage Description	Allows you to enter the reason for accessing the location of the iOS device.
Microphone Usage Description	Allows you to enter the reason for accessing the microphone on the iOS device.
Mute Other Audio Sources	Enable this if you want your Unity application to stop audio from applications running in the background. Disable this if you want audio from background applications to continue playing alongside your Unity application.

Property:	*Function:*
Prepare iOS for Recording	When selected, the microphone recording APIs are initialised. This makes recording latency lower, though on iPhones it re-routes audio output via earphones only.
Force iOS Speakers when Recording	Make the phone output through the internal speakers, even when headphones are plugged in and recording.
Requires Persistent WiFi	Specifies whether the application requires a Wi-Fi connection. iOS maintains the active Wi-Fi connection while the application is running.
Allow downloads over HTTP (nonsecure)	When this option is enabled it will allow you to download content over HTTP. Default and recommended is HTTPS.
Supported URL schemes	A list of supported URL schemes.
Disable HW Statistics	By default, Unity iOS apps send anonymous HW statistics to Unity so we can provide you with aggregated information to help you make decisions as a developer. These stats can be found at http://stats.unity3d.com/. Checking this option disables the sending of these statistics for your app.
Target Device	Which devices are targeted by the game? The options are **iPhone Only**, **iPad Only** and **iPhone + iPad**.
Target SDK	Which SDK is targeted by the game? The options are **iPhone Only**, **Device SDK** and **Simulator SDK**.
Target minimum iOS Version	Defines the minimum version of iOS that the game will work.
Enable ProMotion Support	Enable high frequency refresh rates (120 Hz) on ProMotion displays. Enabling this setting might affect battery life.
Requires ARKit support	Restricts the app to iPhone 6s/iOS 11 or newer devices when publishing to the App Store.
Defer system gestures on edges	Users must swipe twice on the selected edges to enact system gesture.
Hide home button on iPhone X	Hide the home button on iPhone X devices when the app is running.
Render Extra Frame on Pause	Issue an additional frame after the frame when the app is paused. This allows your app to show graphics that indicate the paused state when the app is going into the background.
Behaviour in Background	Specifies what the application should do when the user presses the home button.
- Suspend	This is the standard behaviour; the app is suspended, but not quit.
- Exit	Instead of suspending, the app will quit when the home button is pressed.
- Custom	You can implement your own behaviour with background processing. See an example here.
Architecture	Allows you to select which architecture to target. Universal is recommended default. Some apps that are shipping on high-end devices only might consider selecting the Arm64-only option. Armv7 is for consistency purposes.
- Universal	The recommended option. Supports both architectures.
- Armv7	Support only the older Armv7 architecture.
- Arm64	Support only the newer Arm64 architecture.
- x86_64	Only architecture available for **Simulator SDK**. Supports the x86_64 architecture.
Scripting Define Symbols	Custom compilation flags (see the platform dependent compilation page for details).

Property:	*Function:*
Allow 'unsafe' Code	Enables support for compiling 'unsafe' C# code in a pre-defined assembly (for example, *Assembly-CSharp.dll*). For Assembly Definition Files (*.asmdef*), click on one of your *.asmdef* files and enable the option in the **Inspector** window that appears.
Active Input Handling	Choose to handle input using the Input Manager, the **Input System (Preview)**, or **Both**. If you change this setting, you must restart Unity for the change to take effect.
Optimization	
Prebake Collision__ Meshes|**Should** collision__A collision occurs when the physics engine detects that the colliders of two GameObjects make contact or overlap, when at least one has a rigidbody component and is in motion. More info See in Glossary data be added to meshes at build time?	
Keep Loaded Shaders Alive	Retains shaders in memory for faster access when the player starts the game up.
Preloaded Assets	An array of assets to be loaded when the player starts up.
AOT compilation options	Additional AOT compiler options.
Strip Engine Code	Enable code stripping. (This setting is only available with the IL2CPP scripting backend.)
Script Call Optimization	Optionally disable exception handling for a speed boost at runtime. See iOS Optimization for details.
- Slow and Safe	Full exception handling will occur (with some performance impact on the device when using the Mono scripting backend).
- Fast but no Exceptions	No data provided for exceptions on the device (the game will run faster when using the Mono scripting backend).
Vertex Compression	Select which vertex channels should be compressed. **Compression** can save memory and bandwidth but precision will be lower.
Optimize Mesh__ Data__	Remove any data from meshes that is not required by the material applied to them (tangents, normals, colors, UV).

Note: Be sure to select the correct SDK - if you select *Device*, say, but then target the Simulator in Xcode then the build will fail with a lot of error messages.

API Compatibility Level

You can choose your Mono API compatibility level for all targets. Sometimes a 3rd party .net dll will use things that are outside of the .NET compatibility level that you would like to use. To understand what is going on in such cases, and how to best fix it, get "Reflector" on Windows.

Drag the .NET assemblies for the api compatilibity level in question into reflector. You can find these in Frameworks/Mono/lib/mono/YOURSUBSET/
Also drag in your 3rd party assembly.
Right click your 3rd party assembly, and select "Analyze".
In the analysis report, inspect the "Depends on" section. Anything that the 3rd party assembly depends on, but is not available in the .NET compatibility level of your choice will be highlighted in red there.

Bundle Identifier

The **Bundle Identifier** string must match the provisioning profile of the game you are building. The basic structure of the identifier is **com.CompanyName.GameName**. This structure may vary internationally based on where you live, so always default to the string provided to you by Apple for your Developer Account. Your GameName is set up in your provisioning certificates, that are manageable from the Apple iPhone Developer Center website. Please refer to the Apple iPhone Developer Center website for more information on how this is performed.

Stripping level

Most games don't use all necessary dlls. With the **Strip Engine Code** option enabled, you can strip out unused parts to reduce the size of the built player on iOS devices. If your game is using classes that would normally be stripped out by the option you currently have selected, you'll be presented with a Debug message when you make a build.

Script call optimization

A good development practice on iOS is to never rely on exception handling (either internally or through the use of try/catch blocks). When using the default **Slow and Safe** option, any exceptions that occur on the device will be caught and a stack trace will be provided. When using the **Fast but no Exceptions** option, any exceptions that occur will crash the game, and no stack trace will be provided. In addition, the **AppDomain.UnhandledException** event will be raised to allow project-specific code access to the exception information.

With the Mono scripting backend, the game will run faster since the processor is not diverting power to handle exceptions. There is no performance benefit with the **Fast but no Exceptions** option when using the IL2CPP scripting backend. When releasing your game to the world, it's best to publish with the **Fast but no Exceptions** option.

XR settings

Property:	*Function:*
XR Settings	
Virtual Reality Supported	Enable native VR support for the Unity Editor and your game builds.
Vuforia Augmented Reality Supported	Enable use of the Vuforia Software Development Kit. You must have a Vuforia Software License and agree to the terms of that license before the property becomes enabled.

Incremental builds

The C++ code generated by the IL2CPP scripting backend can be updated incrementally, allowing incremental C++ build systems to compile only the changes source files. This can significantly lower iteration times with the IL2CPP scripting backend.

To use incremental builds, choose the "Append" option after selecting "Build" from the "Build Settings" dialog. The "Replace" option will perform a clean build.

2018–08–16 Page amended with editorial review

Mute Other Audio Sources added in 5.5

iOS Player Settings documentation updated in Unity 2018.1

Allow 'unsafe' Code checkbox added in Unity 2018.1

.NET 4.x runtime added in 2018.1

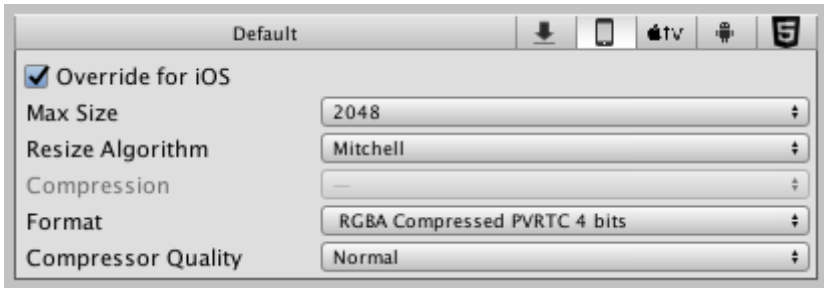
.Net 3.5 scripting runtime deprecated in Unity 2018.3

iOS 2D Texture Overrides

Leave feedback

When you are building for different platforms, you have to think about the resolution of your textures for the target platform, the size and the quality. You can set default options and then override the defaults for a specific platform.

This page details the **Texture Overrides** specific to **iOS**. A description of the general Texture Overrides can be found here.



Property	*Function*
Format	What internal representation is used for the texture. This is a tradeoff between size and quality. In the examples below we show the final size of an in-game texture of 256 by 256 **pixels**:
RGB Compressed PVRTC__ 4 bits__	Compressed RGB texture. This is the most common format for diffuse textures. 4 bits per pixel (32 KB for a 256x256 texture).
RGBA Compressed PVRTC 4 bits	Compressed RGBA texture. This is the main format used for diffuse and specular control textures or diffuse textures with transparency. 4 bits per pixel (32 KB for a 256x256 texture).
RGB Compressed PVRTC 2 bits	Compressed RGB texture. Lower quality format suitable for diffuse textures. 2 bits per pixel (16 KB for a 256x256 texture).
RGBA Compressed PVRTC 2 bits	Compressed RGBA texture. Lower quality format suitable for diffuse & specular control textures. 2 bits per pixel (16 KB for a 256x256 texture).

Property	*Function*
RGB Compressed ASTC 4x4 block	
RGB Compressed ASTC 5x5 block	
RGB Compressed ASTC 6x6 block	Compressed RGB texture. Supported by all Metal, OpenGL ES 3.2 and OpenGL ES 3.1+AEP GPUs, as well as by some OpenGL ES 3.0 GPUs. This compression type uses fixed 128-bit block size, and depending on the pixel block size (4x4 to 12x12), it may utilize 8 to 0.89 bits per pixel. The size of the compressed texture varies from 64 KB for a 256x256 texture (4x4 block, highest quality) to 7.6 KB for a 256x256 texture (12x12 block, highest compression rate).
RGB Compressed ASTC 8x8 block	
RGB Compressed ASTC 10x10 block	
RGB Compressed ASTC 12x12 block	

Property	*Function*
RGBA Compressed ASTC 4x4 block	
RGBA Compressed ASTC 5x5 block	
RGBA Compressed ASTC 6x6 block	Compressed RGBA texture. Supported by all Metal, OpenGL ES 3.2 and OpenGL ES 3.1+AEP GPUs, as well as by some OpenGL ES 3.0 GPUs. This compression type uses fixed 128-bit block size, and depending on the pixel block size (4x4 to 12x12), it may utilize 8 to 0.89 bits per pixel. The size of the compressed texture varies from 64 KB for a 256x256 texture (4x4 block, highest quality) to 7.6 KB for a 256x256 texture (12x12 block, highest compression rate).
RGBA Compressed ASTC 8x8 block	
RGBA Compressed ASTC 10x10 block	
RGBA Compressed ASTC 12x12 block	
RGB Compressed ETC__ 4 bits__	Compressed RGB texture. This is the default **texture format** for Android projects in Unity. ETC_RGB4 is a part of OpenGL ES 2.0 and is supported by all OpenGL ES 2.0 GPUs. It does not support alpha. 4 bits per pixel (32 KB for a 256x256 texture).
RGB Crunched ETC	Compressed RGB texture. Crunch is a lossy **compression** format on top of ETC texture **compression**. Textures are decompressed to ETC_RGB4 by the CPU and then uploaded to the GPU at runtime. Crunch produces smaller textures than by regular ETC_RGB4 **compression**, but with lower quality. Crunch textures can take a long time to compress, but decompression at runtime is very fast. ETC_RGB4 is a part of OpenGL ES 2.0 and is supported by all Metal and OpenGL ES 2.0 GPUs. It does not support alpha. 4 bits per pixel (output size varies depending on the texture - from 1 KB for a 256x256 texture).
RGBA Compressed ETC2 8 bits	Compressed RGBA texture. Supported by all Metal and OpenGL ES 3.0 GPUs. (64 KB for a 256x256 texture).

Property	*Function*
RGBA Crunched ETC2	Compressed RGBA texture. Crunch is a lossy compression format on top of ETC **texture compression**. Textures are decompressed to ETC2_RGBA8 on the CPU and then uploaded to the GPU at runtime. Crunch compression can be used to produce significantly smaller textures than by regular ETC2_RGBA8 compression, but with lower quality. Crunch textures can take a long time to compress, but decompression at runtime is very fast. Supported by all Metal and OpenGL ES 3.0 GPUs. 8 bits per pixel (output size varies depending on the texture - from 1 KB for a 256x256 texture).
RGB 16 bit	65 thousand colors with no alpha. Uses more memory than PVRTC formats, but could be more suitable for **UI** or crisp textures without gradients. 128 KB for a 256x256 texture.
RGB 24 bit	Truecolor but without alpha. 192 KB for a 256x256 texture.
Alpha 8 bit	High quality alpha channel but without any color. 64 KB for a 256x256 texture.
RGBA 16 bit	Low-quality truecolor. Has 16 levels of red, green, blue and alpha. Uses more memory than PVRTC formats, but can be handy if you need exact alpha channel. 128 KB for a 256x256 texture.
RGBA 32 bit	Truecolor with alpha - this is the highest quality. At 256 KB for a 256x256 texture, this one is expensive. Most of the time, **PVRTC** formats offers sufficient quality at a much smaller size.
Compressor quality	Choose Fast for quickest performance, Best for the best image quality and Normal for a balance between the two.

2018–06–14 Page amended with limited editorial review

Crunch compression format updated in Unity 2017.3

Upgrading to 64-bit iOS

Leave feedback

iOS 64-bit support is implemented with the scripting backend called IL2CPP. It runs with your C# code.

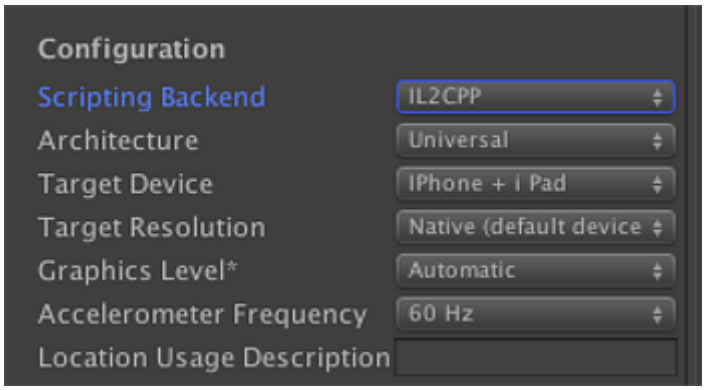
The IL2CPP runtime combines an ahead of time compiler with a virtual machine to convert assemblies to C++ while leveraging standard platform C++ compilers to produce native binaries. The net result is significantly higher performance, platform compatibility and maintainability.

IL2CPP is the only scripting backend to support deploying to ARM 64-bit on **iOS**, and is thus mandatory to deploy to the Apple App Sstore for releasing new apps.

To read more about IL2CPP, see the blog posts The future of scripting in Unity and Apple iOS 64-bit support in Unity.

How to start using IL2CPP on iOS

Select **IL2CPP** in the **Scripting Backend** drop-down menu in Player Settings.



Scripting backend selection

By default it will build for **Universal** architecture (including both *ARM64* and *ARMv7*); if needed you can switch to specific architecture in Player Settings. There are a number of things that should be done before your application is up and running in 64-bits:

You need a 64-bit capable device to test on: all iOS devices with A7 or later chip (currently these are: iPhone 5S, iPad Air, iPad Mini Retina, iPhone 6, iPhone 6 Plus, iPad Mini 3, and all newer devices).
You need all your native **plugins** to be compiled with 64-bit support (or be provided as source code). If you are using a 3rd party plugin, you should contact your plugin vendor to obtain a 64-bit capable and IL2CPP-compatible version of that plugin. At the moment all latest Prime31 plugins should work.
If you are the plugin vendor or just have your own **native plugins** then you need keep couple of things in mind:
IL2CPP is not API-compatible (which is unexposed) with the Mono Runtime API, which means if plugin is using *mono_** functions won't link anymore. The best way to resolve this issue is to switch to a managed delegate/callback approach and pass your managed callbacks to the native side of plugin, and call them back from there when some native data or event arrives.
All plugins that come as precompiled static libraries (.a files) should now include *ARM64* slice.
When doing native code/type conversion keep in mind that pointers and longs are now 64-bit wide and ints are still 32-bit wide.

To minimise generated amount of C++ code IL2CPP is always (even when *Stripping level* is set to **Disabled**) doing some sort of managed code stripping. Sometimes you will need to help it. Look for the link.xml guide in the iOS player size optimization manual.

Troubleshooting

Q: My code is running slower on IL2CPP than on Mono. Why?
A: Make sure you are testing your code performance in Xcode *Release* configuration. If issue still remains, please submit a bug report!
Q: Unity generated Xcode project fails to compile with following or similar error: `Method not found: 'Default constructor not found...ctor() of System.ComponentModel.Int64Converter'.`
A: Deserializers and serializers often reference some types only via .NET Reflection API and in such cases these methods or even classes might be stripped from the project. You can hint managed code stripper that specific class / method is used either via link.xml or via introduction of dummy code that explicitly references it in one of your **scripts**.
Q: Unity generated Xcode project fails to compile with some other error.
A: Please verify that all your plugins support *ARM64* and IL2CPP and if so, please submit a bug report and attach your project (or just the scripting part of it) to the bug report.

2018–06–14 Page amended with limited editorial review

Unity Remote

Leave feedback

Unity Remote is a downloadable app designed to help with Android, **iOS** and tvOS development. The app connects with Unity while you are running your project in Play Mode from the Unity Editor. The visual output from the Editor is sent to the device's screen, and the live inputs are sent back to the running project in Unity. This allows you to get a good impression of how your game really looks and handles on the target device, without the hassle of a full build for each test.

Note: For Unity Remote to work, you need to have the Android SDK on your development machine.

Unity Remote replaces the separate iOS and Android Remote apps used with earlier versions. Those older Remote apps are no longer supported.

Older versions of Unity Remote are still available for use in Legacy projects; see Legacy Unity Remote documentation for more information on these versions.

Device and Feature Support

Unity Remote currently supports Android devices (on Windows and OS X via a USB connection) and iOS devices (iPhone, iPad, iPod touch and Apple TV, through USB on OS X and Windows with iTunes).

The Game View of the running Unity project is duplicated on the device screen, but at a reduced framerate. The following input data from the device is also streamed back to the Editor:

Touch and stylus input
Accelerometer
Gyroscope
Device **camera** streams
Compass
GPS
Joystick names and input

Note that the Remote app simply shows the visual output on the device and takes input from it. The game's actual processing is still done by the Editor on the desktop machine - so its performance is not a perfect reflection of the built app.

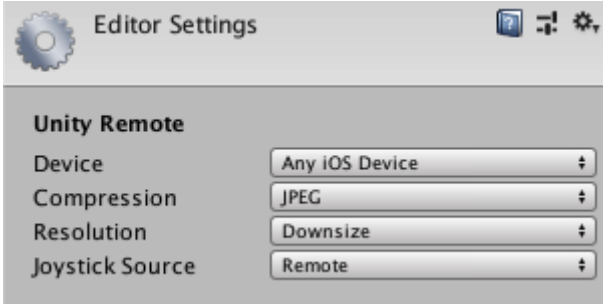
Obtaining and Using Unity Remote

You can download Unity Remote for free in the form of a Unity project that you build yourself, or as a pre-built app from the device's app store:

Unity Project (requires custom building) from the Asset Store
Android app from Google Play
iOS and tvOS apps from the App Store

Once you've downloaded the app, install and run it on your device and connect the device to your computer using a USB cable.

To enable Unity to work with your device, open the Editor Settings in Unity (menu: **Edit > Project Settings > Editor**) and select the device to use from the **Unity Remote** section:



Note: To specify the location of your Android SDK, go to **Unity > Preferences > External Tools**.

Click the Play button in the Editor to see your game appear on the device and in the Unity game window as Unity connects to the Remote app. While the game plays, input from the device (accelerometer, etc.) is sent to your **scripts** as if they were running on the device itself.

Troubleshooting

I have more than one device plugged in, but only one of them works with Unity

Unity Remote doesn't support multiple connected Android devices, and to resolve this, it will automatically pick the first device it finds. However, it is fine to have multiple iOS/tvOS devices and one Android device connected at the same time, since you can select which one to use from the Editor Settings (menu: **Edit > Project Settings > Editor**).

I'm getting really poor graphics quality when running my game in Unity Remote

When you use Unity Remote the game actually runs in the Editor, while its visual content is streamed to the target device. Since the bandwidth between the Editor and the device is limited, the stream must be compressed heavily for transmission. This **compression** inevitably reduces the image quality.

In the **Unity Remote** section of the Editor settings (menu: **Edit > Project Settings > Editor**) you can switch the compression method between JPEG and PNG. PNG compression is "lossless" (so the image quality doesn't degrade) but uses more bandwidth than JPEG. A downsized image has lower bandwidth requirements than one at full resolution. By changing these settings, you can trade image accuracy off against framerate as necessary. When using a lower resolution, be aware that Unity Remote downsizes images for better performance.

Bear in mind that Unity Remote is only really intended to give a quick approximate check of how your game will look and feel when running on the device. Make sure that you occasionally do a full build and test the "real" app.

2018–06–14 Page amended with limited editorial review

iOS Scripting

Leave feedback

Most features of the **iOS** devices are exposed through the Input and Handheld classes. For cross-platform projects, **UNITY_IPHONE** is defined for conditionally compiling iOS-specific C# code.

iOS Game Controller support

Leave feedback

Unity supports a standardized **Game Controller** Input API provided by Apple as part of the standard Unity Input API.

Detecting attached Game Controllers

Calling `Input.GetJoystickNames` will enumerate the names of all attached controllers. Names follow the pattern: `[$profile_type,$connection_type] joystick $number by $model`. $profile_type might be either "basic" or "extended", $connection_type is either "wired" or "wireless". It could be used to detect the kind of connected controller. It is worth re-checking this list every few seconds to detect if controllers have been attached or detached. Here's a C# example:

```
private bool connected = false;

IEnumerator CheckForControllers() {
    while (true) {
        var controllers = Input.GetJoystickNames();
        if (!connected && controllers.Length > 0) {
            connected = true;
            Debug.Log("Connected");
        } else if (connected && controllers.Length == 0) {
            connected = false;
            Debug.Log("Disconnected");
        }
        yield return new WaitForSeconds(1f);
    }
}

void Awake() {
    StartCoroutine(CheckForControllers());
}
```

When controllers are detected you can either hide on-screen touch controls or amend them to supplement controller input. The next task is to check for Game Controller input.

Handling input

The actual input scheme is highly dependent on the type of game you are developing. But it's essentially about reading axes and button states. Take the following 2D game stage as a simple example:

The player controls the bean character, which can move right or left, jump and fire at the bad guys. By default, the Unity Input "Horizontal" axis is mapped to basic profile game controller D-pad and the left analog stick on extended profile controllers. So the code used to move the character back and forth is pretty simple:

```
float h = Input.GetAxis("Horizontal");
if (h * rigidbody2D.velocity.x < maxSpeed)
    rigidbody2D.AddForce(Vector2.right * h * moveForce);
```

You can set up jump and fire actions in Unity's **Input Manager**.

Access it from the Unity editor menu as follows: **Edit** > **Project Settings** > **Input**.

Pick joystick button "A" for the "Jump" action and "X" for "Fire".

Open the following actions in the Unity Input Manager and specify "joystick button 14" for "Jump" and "joystick button 15" for "Fire".

The code handling then looks like this:

```
if (Input.GetButtonDown("Jump") && grounded) {
    rigidbody2D.AddForce(new Vector2(0f, jumpForce));
}

if (Input.GetButtonDown("Fire")) {
    Rigidbody2D bulletInstance = Instantiate(rocket, transform.position, Quatern
    bulletInstance.velocity = new Vector2(speed, 0);
}
```

The following cheatsheet should help you map controller input in the Unity Input Manager:

Name	KeyCode	Axis
A	joystick button 14	joystick axis 14
B	joystick button 13	joystick axis 13
X	joystick button 15	joystick axis 15
Y	joystick button 12	joystick axis 12
Left Stick	N/A	Axis 1 (X) - Horizontal, Axis 2 (Y) - Vertical
Right Stick	N/A	Axis 3 - Horizontal, Axis 4 - Vertical
D-pad Up	joystick button 4	Basic profile only: Axis 2 (Y)
D-pad Right	joystick button 5	Basic profile only: Axis 1 (X)
D-pad Down	joystick button 6	Basic profile only: Axis 2 (Y)
D-pad Left	joystick button 7	Basic profile only: Axis 1 (X)
Pause	joystick button 0	N/A
L1/R1	joystick button 8 / joystick button 9	joystick axis 8 / joystick axis 9
L2/R2	joystick button 10 / joystick button 11	joystick axis 10 / joystick axis 11

Final notes on Game Controller API support

Unity only includes the Game Controller framework in the project if a script in the project references `Input.GetJoystickNames`. Unity **iOS** Runtime loads the framework dynamically, if it is available.

Apple documentation explicitly states that controller input must be optional and your game should be playable without them.

2018–06–14 Page amended with limited editorial review

Advanced Unity Mobile Scripting

Leave feedback

Device Properties

There are a number of device-specific properties that you can access. See the script reference pages for SystemInfo.deviceUniqueIdentifier, SystemInfo.deviceName, SystemInfo.deviceModel and SystemInfo.operatingSystem.

Anti-Piracy Check

Pirates will often hack an application (by removing AppStore DRM protection) and then redistribute it for free. Unity comes with an anti-piracy check which allows you to determine if your application was altered **after** it was submitted to the AppStore.

You can check if your application is genuine (not hacked) with the Application.genuine property. If this property returns **false** then you might notify user that they are using a hacked application or maybe disable access to some functions of your application.

Note: Application.genuineCheckAvailable should be used along with **Application.genuine** to verify that application integrity can actually be confirmed. Accessing the Application.genuine property is a fairly expensive operation and so you shouldn't do it during frame updates or other time-critical code.

Vibration Support

You can trigger a vibration by calling Handheld.Vibrate. However, devices lacking vibration hardware will just ignore this call.

Activity Indicator

Mobile OSes have built-in activity indicators, that you can use during slow operations. Please check Handheld.StartActivityIndicator docs for sample usage.

Screen Orientation

Unity **iOS**/Android allows you to control current screen orientation. Detecting a change in orientation or forcing some specific orientation can be useful if you want to create game behaviors depending on how the user is holding the device.

You can retrieve device orientation by accessing the Screen.orientation property. Orientation can be one of the following:

Portrait	The device is in portrait mode, with the device held upright and the home button at the bottom.
PortraitUpsideDown	The device is in portrait mode but upside down, with the device held upright and the home button at the top.
LandscapeLeft	The device is in landscape mode, with the device held upright and the home button on the right side.
LandscapeRight	The device is in landscape mode, with the device held upright and the home button on the left side.

You can control screen orientation by setting Screen.orientation to one of those, or to ScreenOrientation.AutoRotation. When you enable auto-rotation, you can still disable some orientation on a case by case basis. See the script reference pages for Screen.autorotateToPortrait, Screen.autorotateToPortraitUpsideDown, Screen.autorotateToLandscapeLeft and Screen.autorotateToLandscapeRight

---

2018–06–14 Page amended with limited editorial review

Advanced iOS scripting

Determining Device Generation

Different device generations support different functionality and have widely varying performance. You should query the device's generation and decide which functionality should be disabled to compensate for slower devices. You can find the device generation from the iOS.DeviceGeneration property.

More information about different device generations, performance and supported functionality can be found in our iPhone Hardware Guide.

Optimizing Performance on iOS

Leave feedback

This section covers optimizations which are unique to **iOS** devices.

iOS-specific optimizations

Leave feedback

This page details optimizations that are unique to **iOS** deployment.

Script call optimization

Most of the functions in the **UnityEngine** namespace are implemented in C/C++. Calling a C/C++ function from a Mono script involves a performance overhead, so you can save about 1 to 4 milliseconds per frame using iOS Script Call optimization.

To access iOS Script Call optimization, navigate to the Player Settings window (menu: **Edit** > **Project Settings** > **Player**) and select the iOS icon (shown below).

iOS Icon - select this to access the iOS settings in the Player Settings window

Locate the **Script Call Optimization** setting in the *_Other Settings* section.

The options for this setting are:-

Slow and Safe - the default Mono internal call handling with exception support.
Fast but no exceptions - a faster implementation of Mono internal call handling. However, this doesn't support exceptions and so should be used with caution. An app that doesn't explicitly handle exceptions (and doesn't need to deal with them gracefully) is an ideal candidate for this option.

Note: There is no performance benefit when using the **IL2CPP scripting backend**. However, we recommend using **Fast but no exceptions** on release builds to avoid undefined behaviour.

Setting the frame rate

Unity iOS allows you to change the frequency with which your application will try to execute its **rendering** loop, which is set to 30 **frames per second** by default. You can lower this number to save battery power but of course this saving will come at the expense of frame updates. Conversely, you can increase the framerate to give the **rendering** priority over other activities such as touch input and accelerometer processing. You will need to experiment with your choice of framerate to determine how it affects gameplay in your case.

If your application involves heavy computation or rendering and can maintain only 15 frames per second, say, then setting the desired frame rate higher than fifteen wouldn't give any extra performance. The application has to be optimized sufficiently to allow for a higher framerate.

To set the desired frame rate, change Application.targetFrameRate.

Tuning accelerometer processing frequency

If accelerometer input is processed too frequently then the overall performance of your game may suffer as a result. By default, a Unity iOS application will sample the accelerometer 60 times per second. You may see some performance benefit by reducing the accelerometer sampling frequency and it can even be set to zero for games that don't use accelerometer input. You can change the accelerometer frequency from the **Other Settings** panel in the iOS Player Settings.

2018–06–14 Page amended with limited editorial review

Measuring performance with the built-in profiler

Leave feedback

Unity **iOS** and Android contain a built-in **profiler**. The built-in **profiler** emits console messages from the game running on device. These messages are written every 30 seconds and will provide insight into how the game is running. Understanding what these messages mean is not always easy, but as a minimum, you should quickly be able to determine if your game is CPU or GPU bound, and if CPU bound whether it's script code, or perhaps Mono garbage collection that is slowing you down. See later in this page to learn how to configure the built-in **profiler**.

What the profiler tells you

Here's an example of the built-in profiler's output.

```
iPhone/iPad Unity internal profiler stats:
cpu-player> min: 9.8 max: 24.0 avg: 16.3
cpu-ogles-drv> min: 1.8 max: 8.2 avg: 4.3
cpu-waits-gpu> min: 0.8 max: 1.2 avg: 0.9
cpu-present> min: 1.2 max: 3.9 avg: 1.6
frametime> min: 31.9 max: 37.8 avg: 34.1
draw-call #> min: 4 max: 9 avg: 6 | batched: 10
tris #> min: 3590 max: 4561 avg: 3871 | batched: 3572
verts #> min: 1940 max: 2487 avg: 2104 | batched: 1900
player-detail> physx: 1.2 animation: 1.2 culling: 0.5 skinning: 0.0 batching: 0.2
mono-scripts> update: 0.5 fixedUpdate: 0.0 coroutines: 0.0
mono-memory> used heap: 233472 allocated heap: 548864 max number of collections: 1
```

All times are measured in milliseconds per frame. You can see the minimum, maximum and average times over the last thirty frames.

General CPU activity

Property:	Function:
cpu-player	Displays the time your game spends executing code inside the Unity engine and executing **scripts** on the CPU.
cpu-ogles-drv	Displays the time spent executing OpenGL ES driver code on the CPU. Many factors like the number of draw calls, number of internal **rendering** state changes, the **rendering** pipeline setup and even the number of processed vertices can have an effect on the driver stats.
cpu-waits-gpu	Displays the time the CPU is idle while waiting for the GPU to finish rendering. If this number exceeds 2–3 milliseconds then your application is most probably fillrate/GPU processing bound. If this value is too small then the profile skips displaying the value.
msaa-resolve	The time taken to apply anti-aliasing.

Property:	Function:
cpu-present	The amount of time spent executing the presentRenderbuffer command in OpenGL ES.
frametime	Represents the overall time of a game frame. Note that iOS hardware is always locked at a 60Hz refresh rate, so you will always get multiples times of 16.7ms (1000ms/60Hz = 16.7ms).

Rendering statistics

Property:	Function:
tris #	Total number of triangles sent for rendering.
verts #	Total number of vertices sent for rendering. You should keep this number below 10000 if you use only static geometry but if you have lots of skinned geometry then you should keep it much lower.
batched	Number of draw-calls, triangles and vertices which were automatically batched by the engine. Comparing these numbers with draw-call and triangle totals will give you an idea how well is your **scene** prepared for batching. Share as many materials as possible among your objects to improve batching.

Detailed Unity Player statistics

The **player-detail** section provides a detailed breakdown of what is happening inside the engine:

Property:	Function:
physx	Time spent on physics.
animation	Time spent animating bones.
culling	Time spent culling objects outside the **camera** frustum.
skinning	Time spent applying animations to skinned meshes.
batching	Time spent batching geometry. Batching dynamic geometry is considerably more expensive than batching static geometry.
render	Time spent rendering visible objects.
fixed-update-count	Minimum and maximum number of FixedUpdates executed during this frame. Too many FixedUpdates will deteriorate performance considerably.

Detailed script statistics

The **mono-scripts** section provides a detailed breakdown of the time spent executing code in the Mono runtime:

Property:	Function:
update	Total time spent executing all Update() functions in scripts.
fixedUpdate	Total time spent executing all FixedUpdate() functions in scripts.
coroutines	Time spent inside script coroutines.

Detailed statistics on memory allocated by scripts

The **mono-memory** section gives you an idea of how memory is being managed by the Mono garbage collector:

Property:	Function:

Property:	Function:
allocated heap	Total amount of memory available for allocations. A garbage collection will be triggered if there is not enough memory left in the heap for a given allocation. If there is still not enough free memory even after the collection then the allocated heap will grow in size.
used heap	The portion of the **allocated heap** which is currently used up by objects. Every time you create a new class instance (not a struct) this number will grow until the next garbage collection.
max number of collections	Number of garbage collection passes during the last 30 frames.
collection total duration	Total time (in milliseconds) of all garbage collection passes that have happened during the last 30 frames.

Configuration

On iOS, it's disabled by default. To enable it, open the Unity-generated XCode project, select the `InternalProfiler.h` file, and change the line

```
#define ENABLE_INTERNAL_PROFILER 0
```

to

```
#define ENABLE_INTERNAL_PROFILER 1
```

Select **View > Debug Area > Activate Console** in the XCode menu to display the output console (GDB) and then run your project. Unity will output statistics to the console window every thirty frames.

To enable it on Android, click the **Enable Internal Profiler** (Deprecated) checkbox in PlayerSettings (**Edit** > **Project Settings** > **Player**). Make sure **Development Build** is checked in the **Build Settings** when building, and the statistics should show up in logcat when run on the device. To view logcat, you need **adb** or the Android Debug Bridge. Once you have that, simply run the shell command **adb logcat**.

2018–06–14 Page amended with limited editorial review

Optimizing the size of the built iOS Player

Leave feedback

The two main ways of reducing the size of the Player are by making a proper **Release build** within Xcode and by changing the **Stripping Level** within Unity.

Building for distribution

It is expected that final release builds are made using the Xcode command **Product** > **Archive**. Using this command ensures that build is made with release configuration **and** all the debug symbols are stripped. After issuing this command, Xcode switches to the Organizer window **Archives** tab. For guidelines on how to calculate app size and other size-reducing tips, see Apple's Technical Q&A on Reducing the size of my App.

Note: We recommend you have some small extra margin for error when aiming for an over-the-air download limit (which currently is 150MB).

iOS stripping level

Activate the size optimizations for Mono **scripting backend** builds by stripping work in the following way:

Strip assemblies level: the **scripts**' bytecode is analyzed so that classes and methods that are not referenced from the **scripts** can be removed from the DLLs and thereby excluded from the **AOT compilation** phase. This optimization reduces the size of the main binary and accompanying DLLs and is safe as long as no reflection is used.

Strip ByteCode level: any .NET DLLs (stored in the Data folder) are stripped down to metadata only. This is possible because all the code is already precompiled during the AOT phase and linked into the main binary.

Use micro mscorlib level: a special, smaller version of mscorlib is used. Some **components** are removed from this library, for example, Security, Reflection.Emit, Remoting, non Gregorian calendars, **etc**. Also, interdependencies between internal **components** are minimized. This optimization reduces the main binary and mscorlib.dll size but it is not compatible with some System and System.Xml assembly classes, so use it with care.

These levels are cumulative, so level 3 optimization implicitly includes levels 2 and 1, while level 2 optimization includes level 1.

Note that **Micro mscorlib** is a heavily stripped-down version of the core library. Only those items that are required by the Mono runtime in Unity remain. Best practice for using micro mscorlib is not to use any classes or other features of .NET that are not required by your application. GUIDs are a good example of something you could omit; they can easily be replaced with custom made pseudo GUIDs and doing this would result in better performance and app size.

Stripping with IL2CPP

Refer to documentation on managed bytecode stripping with IL2CPP for more information

Note: Tt can sometimes be difficult to determine which classes are getting stripped in error even though the application requires them. You can often get useful information about this by running the stripped application on the simulator and checking the Xcode console for error messages.

Simple checklist for making your distribution as small as possible

Minimize your **assets**: enable **compression** for textures and reduce their resolution as far as possible. Also, minimize the number of uncompressed sounds. There are some additional tips for file size reduction here.
On Mono, set the **iOS Stripping Level** to **Use micro mscorlib**, or on **IL2CPP** enable **Strip Engine Code**.
Set the script call optimization level to **Fast but no exceptions**.
Don't use anything that lives in System.dll or System.Xml.dll in your code. These libraries are **not** compatible with micro mscorlib.
Remove unnecessary code dependencies.
Set the API Compatibility Level to **.Net 2.0 subset**. Note that .Net 2.0 subset has limited compatibility with other libraries.
Avoid generic containers in combination with value types, including structs.

How small can an app be made with Unity?

An empty project would take less than 22 MB in the App Store if all the size optimizations were turned off. With code stripping, the empty **scene** with just the main **camera** can be reduced to less than 12 MB in the App Store (zipped and DRM attached).

Why did my app increase in size after being released to the App Store?

When publishing your app, Apple App Store service first encrypts the binary file and then compresses it via zip. Encryption increases "randomness' of the code segment and thus makes it worse for compression. Check "Building for distribution" chapter above how to estimate App Store size before submission.

2018–06–14 Page amended with limited editorial review

2017–14–06 - Upated Stripping with IL2CPP section

Optimizing Physics Performance

Leave feedback

The NVIDIA PhysX **physics engine** used by Unity is available on **iOS** but the performance limits of the hardware will be reached more easily on mobile platforms than desktops.

Here are some tips for tuning physics to get better performance on iOS:-

You can adjust the **Fixed Timestep** setting (in the Time manager) to reduce the time spent on physics updates. Increasing the timestep will reduce the CPU overhead at the expense of the accuracy of the physics. Often, lower accuracy is an acceptable tradeoff for increased speed.
Set the **Maximum Allowed Timestep** in the Time manager in the 8–10fps range to cap the time spent on physics in the worst case scenario.
Mesh colliders have a much higher performance overhead than primitive colliders, so use them sparingly. It is often possible to approximate the shape of a **mesh** by using child objects with primitive colliders. The child colliders will be controlled collectively as a single compound **collider** by the **rigidbody** on the parent.
While **wheel colliders** are not strictly colliders in the sense of solid objects, they nonetheless have a high CPU overhead.
The total amount of physics calculation depends on the number of non-sleeping rigid bodies and colliders in the **scene** and the complexity of the colliders. You can keep track of how many physics objects there are in the **scene** using the internal profiler.

Building Plugins for iOS

Leave feedback

This page describes Native Code Plugins for the **iOS** platform.

Building an application with a native plugin for iOS

Define your extern method in the C# file as follows:

```
[DllImport ("__Internal")]
private static extern float FooPluginFunction();
```

Set the editor to the iOS build target.

Add your native code source files to the generated Xcode project's *Classes* folder (this folder is not overwritten when the project is updated, but don't forget to backup your native code).

If you are using C++ (.cpp) or Objective-C++ (.mm) to implement the **plugin** you must ensure the functions are declared with C linkage to avoid name mangling issues.

```
extern "C" {
  float FooPluginFunction();
}
```

Plugins written in C or Objective-C do not need this since these languages do not use name-mangling.

Using your plugin from C#

iOS native plugins can be called only when deployed on the actual device, so it is recommended to wrap all native code methods with an additional C# code layer. This code should check Application.platform and call native methods only when the app is running on the device; dummy values can be returned when the app runs in the Editor. See the Bonjour browser sample application for an example.

Calling C# back from native code

Unity iOS supports limited native-to-managed callback functionality via *UnitySendMessage*:

```
UnitySendMessage("GameObjectName1", "MethodName1", "Message to send");
```

This function has three parameters: the name of the target **GameObject**, the script method to call on that object and the message string to pass to the called method.

Known limitations:

Only script methods that correspond to the following signature can be called from native code:
`function MethodName(message:string)`
Calls to *UnitySendMessage* are asynchronous and have a delay of one frame.

Automated plugin integration

Unity iOS supports automated plugin integration in a limited way. All files with extensions **.a**,**.m**,**.mm**,**.c**,**.cpp** located in the *Assets\Plugins\iOS* folder will be merged into the generated Xcode project automatically. However, merging is done by symlinking files from *Assets\Plugins\iOS* to the final destination, which might affect some workflows. The **.h** files are not included in the Xcode project tree, but they appear on the destination file system, thus allowing compilation of .m/.mm/.c/.cpp files.

Note: Subfolders are currently not supported.

iOS Tips

Managed-to-unmanaged calls are quite processor intensive on iOS. Try to avoid calling multiple native methods per frame.
As mentioned above, wrap your native methods with an additional C# layer that calls native code on the device and returns dummy values in the Editor.
String values returned from a native method should be UTF–8 encoded and allocated on the heap. Mono marshalling calls are free for strings like this.
As mentioned above, the Xcode project's *Classes* folder is a good place to store your native code because it is not overwritten when the project is updated.
Another good place for storing native code is the *Assets* folder or one of its subfolders. Just add references from the Xcode project to the native code files: right click on the *Classes* subfolder and choose **Add** > **Existing files**.

Examples

Bonjour Browser Sample

A simple example of the use of a native code plugin can be found here

This sample demonstrates how objective-C code can be invoked from a Unity iOS application. This application implements a very simple Bonjour client. The application consists of a Unity iOS project (*Plugins\Bonjour.cs* is the C# interface to the native code, while *BonjourTest.cs* is the script that implements the application logic) and native code (*Assets\Code*) that should be added to the built Xcode project.

2018–06–14 Page amended with limited editorial review

Preparing your application for In-App Purchases (IAP)

Leave feedback

This chapter does **not** describe how to integrate your game with Apple's *StoreKit* API. It is assumed that you already have integration with StoreKit via a native code plugin.

The Apple StoreKit documentation defines four kinds of products that can be sold via the *In-App Purchase* process:

Content
Functionality
Services
Subscriptions

This chapter covers the first case only and focuses mainly on the downloadable content concept. AssetBundles are recommended for implementing downloadable content in Unity and both the creation and runtime usage of AssetBundles will be covered here.

Exporting your assets for use on iOS

It is sometimes useful to maintain separate projects for your main application and the downloadable AssetBundles that it will use. However, you should note that all *scripts* referenced by objects in the AssetBundle must be present in the main game executable. The project you use to create the AssetBundle must have **iOS** as the selected build target since the content of AssetBundle files is not compatible between **iOS** and other platforms.

AssetBundles are created using editor **scripts** - a simple example is given below:

```
using UnityEngine;
using UnityEditor;

public class ExportBundle : MonoBehaviour {
    [MenuItem ("Assets/Build AssetBundle From Selection - Track dependencies")]
    static void DoExport() {
        string str = EditorUtility.SaveFilePanel("Save Bundle...", Application.dataPath, Selec
        if (str.Length != 0) {
            BuildPipeline.BuildAssetBundle(Selection.activeObject, Selection.objects, str, Bui
        }
    }
}
```

You should save this code in a file called *ExportBundle* and place it inside a folder called *Editor* (you can just create this if it isn't already present in the project). The script will add a menu item entitled *Build AssetBundle From Selection - Track dependencies* on the *Assets* menu in the editor.

The content you want to include in the bundle should be prepared in the form of prefabs. Select a prefab in the Project view and then select *Assets* > *Build AssetBundle From Selection - Track dependencies* (i.e. the menu item added by the ExportBundle script). This command will bring up a save dialog where you can choose the name and location of your AssetBundle file.

Downloading your assets on iOS

Note: Apple may change the folder locations where you are permitted to write data. Always check the latest Apple documentation to be sure your application will be compliant. The following advice was correct as of early 2018.

AssetBundles can be downloaded using the WWW class and once transfer is complete, the enclosed assets can be accessed. The recommended way to download an AssetBundle is to use LoadFromCacheOrDownload as shown in the following sample:

```
IEnumerator GetAssetBundle() {
    WWW download;
    string url = "http://somehost/somepath/someassetbundle.assetbundle";

    while (!Caching.ready)
        yield return null;

    download = WWW.LoadFromCacheOrDownload(url, 0);

    yield return download;

    AssetBundle assetBundle = download.assetBundle;
    if (assetBundle != null) {
        // Alternatively you can also load an asset by name (assetBundle.Load("my asset name")
        Object go = assetBundle.mainAsset;

        if (go != null)
            Instantiate(go);
        else
            Debug.Log("Couldn't load resource");
    } else {
        Debug.Log("Couldn't load resource");
    }
}
```

The downloaded asset bundle files are stored in the *Library* folder of the iOS application sandbox and have the *No Backup* flag set on them. This means the OS won't delete these files accidentally and these files won't be backed up to iCloud.

If you need to choose exactly where the AssetBundle file is stored, you can use a standard WWW download (i.e. just use the constructor instead of LoadFromCacheOrDownload) and then save the downloaded data on disk using the .NET file API. You can save required files to the Application.temporaryCachePath folder (stored in *Library/Caches* which is regularly "cleaned out" by the OS) or the Application.persistentDataPath folder (stored in *Documents* which is not cleaned out by the OS). You should set the *No Backup* flag on these files with iOS.Device.SetNoBackupFlag to prevent them being backed up to iCloud.

Note: If you don't set the *No Backup* flag, your app may be rejected when you submit it to the App Store.

You can access saved files by creating a WWW object with a URL in the form `file:///pathtoyourapplication/Library/savedassetbundle.assetbundle`:-

```
string cachedAssetBundle = Application.temporaryCachePath + "/savedassetbundle.assetbundle";
System.IO.FileStream cache = new System.IO.FileStream(cachedAssetBundle, System.IO.FileMode.Cr
cache.Write(download.bytes, 0, download.bytes.Length);
cache.Close();

iOS.Device.SetNoBackupFlag(cachedAssetBundle);

Debug.Log("Cache saved: " + cachedAssetBundle);
```

Note: You can test the reading of stored files in the Documents folder if you enable *file sharing* (setting `UIFileSharingEnabled` to true in your `Info.plist` allows you to access the *Documents* folder from iTunes). Note that the contents of the Documents folder are cached to iCloud, so you should not use this location to store AssetBundles in the final build to be shipped. See File System Basics in the Apple iOS documentation for further details.

2018–06–14 Page amended with limited editorial review

Customising WWW requests on iOS

Leave feedback

You can use a **plugin** to replace the default code that Unity uses when handling requests with the WWW class. You might do this, for example, to change the default caching behaviour, which caches on the basis of the URL alone.

Example

The following code (described in more detail below) disables the default caching method and adds a “secret” header field to the request in addition to the headers specified from the game’s script code.

```
// Code placed at Plugins/iOS/CustomConnection.mm within the Assets folder.

#include "Unity/WWWConnection.h"

@interface UnityWWWCustomRequestProvider : UnityWWWRequestDefaultProvider
{
}
+ (NSMutableURLRequest*)allocRequestForHTTPMethod:(NSString*)method url:(NSU
@end

@implementation UnityWWWCustomRequestProvider
+ (NSMutableURLRequest*)allocRequestForHTTPMethod:(NSString*)method url:(NSU
{
    NSMutableURLRequest* request = [super allocRequestForHTTPMethod:method u

    // let's pretend for security reasons we dont want ANY cache nor cookies
    request.cachePolicy = NSURLRequestReloadIgnoringLocalCacheData;
    [request setHTTPShouldHandleCookies:NO];

    // let's pretend we want special secret info in header
    [request setValue:@"123456789"forHTTPHeaderField:@"Secret"];

    return request;
}
@end

@interface UnityWWWCustomConnectionDelegate : UnityWWWConnectionDelegate
{
}
@end

@implementation UnityWWWCustomConnectionDelegate
- (NSCachedURLResponse*)connection:(NSURLConnection*)connection willCacheRes
{
    // we dont want caching
```

```
        return nil;
    }
    - (void)connection:(NSURLConnection*)connection didReceiveResponse:(NSURLRes
    {
        // let's just print something here for test
        [super connection:connection didReceiveResponse:response];
        if([response isMemberOfClass:[NSHTTPURLResponse class]])
            ::printf_console("We've got response with status: %d\n", [(NSHTTPURL
    }
    @end

    IMPL_WWW_DELEGATE_SUBCLASS(UnityWWWCustomConnectionDelegate);
    IMPL_WWW_REQUEST_PROVIDER(UnityWWWCustomRequestProvider);
```

The code breaks down as follows. Firstly, we create a subclass of UnityWWWRequestDefaultProvider to provide a modified NSURLRequest object for the connection. Another option is simply to implement the UnityWWWRequestProvider protocol in the class from scratch. In this case, however, we want to make use of Unity's existing code.

```
    @interface UnityWWWCustomRequestProvider : UnityWWWRequestDefaultProvider
    {
    }
    + (NSMutableURLRequest*)allocRequestForHTTPMethod:(NSString*)method url:(NSU
    @end
```

The "action" happens in `allocRequestForHTTPMethod` which starts by calling the same method on the superclass:

```
    @implementation UnityWWWCustomRequestProvider
    + (NSMutableURLRequest*)allocRequestForHTTPMethod:(NSString*)method url:(NSU
    {
        NSMutableURLRequest* request = [super allocRequestForHTTPMethod:method u
```

From there, we disable the default data caching done by **iOS** and also disable cookies:

```
    request.cachePolicy = NSURLRequestReloadIgnoringLocalCacheData;
    [request setHTTPShouldHandleCookies:NO];
```

Next, we add a header field called “Secret” that provides some additional data. In a real game, this might be a value checked by the server to verify the source of the connection.

```
[request setValue:@"123456789"forHTTPHeaderField:@"Secret"];
```

Having created the custom request handling class, we also need to create a subclass of `UnityWWWConnectionDelegate` to customise the handling of the connection:

```
@interface UnityWWWCustomConnectionDelegate : UnityWWWConnectionDelegate
{
}
@end
```

We can disable data caching by returning a nil value from the `connection:willCacheResponse:` method:

```
- (NSCachedURLResponse*)connection:(NSURLConnection*)connection willCacheRes
{
    // we dont want caching
    return nil;
}
```

We can handle the connection itself from `connection:didReceiveResponse:`, which should call the same method on the superclass to actually get the data. Here, we simply print a message to the console when the connection happens but you could implement anything you like here:

```
- (void)connection:(NSURLConnection *)connection didReceiveResponse:(NSURLRe
{
    // let's just print something here for test
    [super connection:connection didReceiveResponse:response];
    if([response isMemberOfClass:[NSHTTPURLResponse class]])
        ::printf_console("We've got response with status: %d\n", [(NSHTTPURL
}
```

Finally, we register our new connection delegate and request provider with Unity so they will be called whenever a WWW request is issued from a script:

```
IMPL_WWW_DELEGATE_SUBCLASS(UnityWWWCustomConnectionDelegate);
IMPL_WWW_REQUEST_PROVIDER(UnityWWWCustomRequestProvider);
```

The custom code is transparent to the use of the WWW class within Unity - no extra code needs to be added to the C# script.

2018–06–14 Page amended with limited editorial review

App thinning

Leave feedback

The Apple store and iOS optimize the installation of iOS, tvOS, and watchOS apps by tailoring app delivery to the capabilities of a user's particular device. This optimization is called app thinning. App thinning lets you create apps that use the most device features, occupy minimum disk space, and accommodate future updates that can be applied by Apple. Learn more about this optimization process on Apple's Developer Library page on App Thinning.

This chapter describes the two major **components** of app thinning that can be implemented in Unity:

On-demand resources (ODR)
App slicing

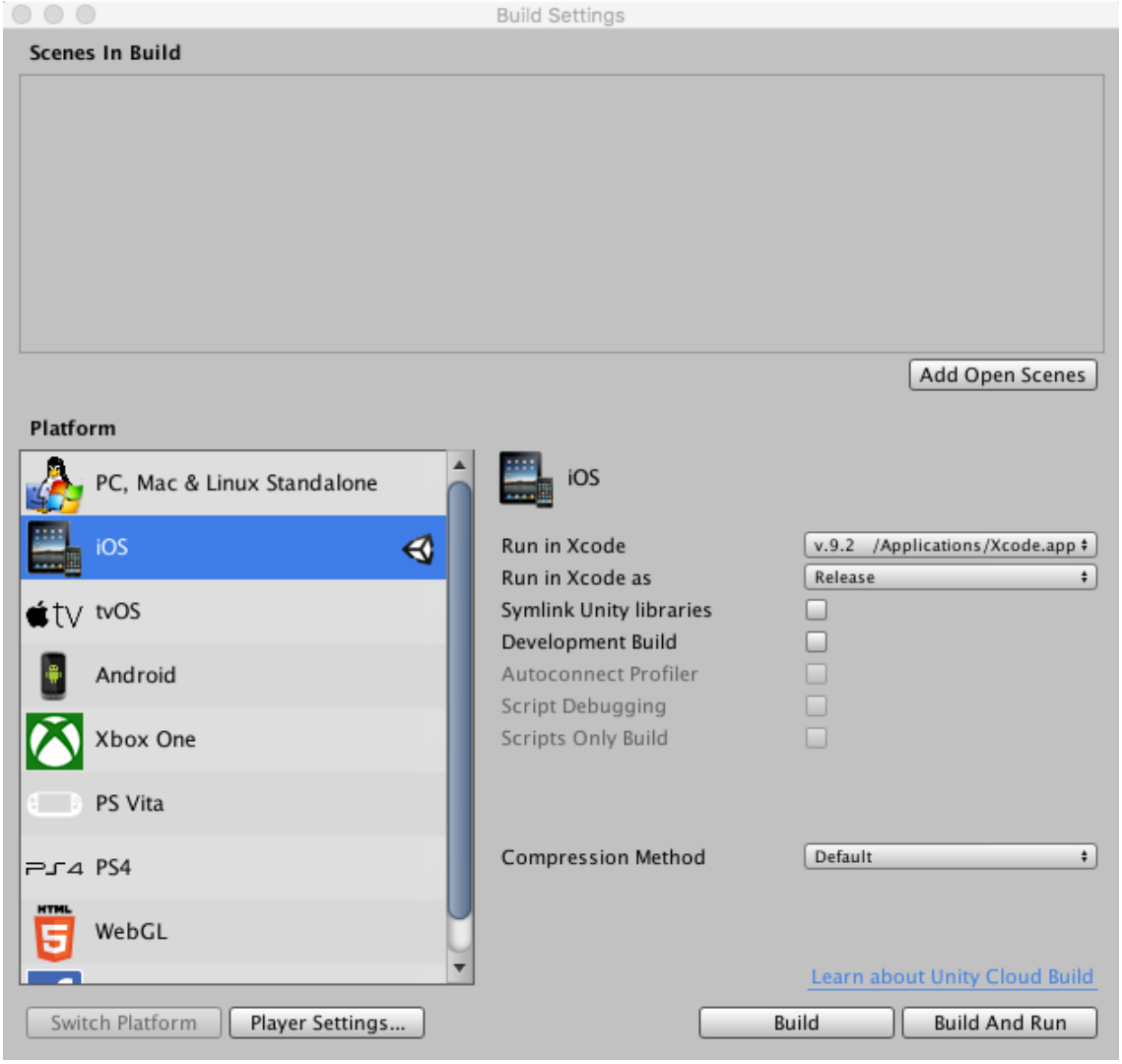
On-demand resources

On-demand resources (ODR) is a feature available for the iOS and tvOS platforms, from version 9.0 of iOS and tvOS onwards. It allows you to reduce the size of your application by separating the core **Assets** (those that are needed from application startup) from Assets which may be optional, or which appear in later levels of your game. These additional Assets are called AssetBundles. They're available across all Unity build targets, but extra steps must be taken to enable them to be hosted via the App Store.

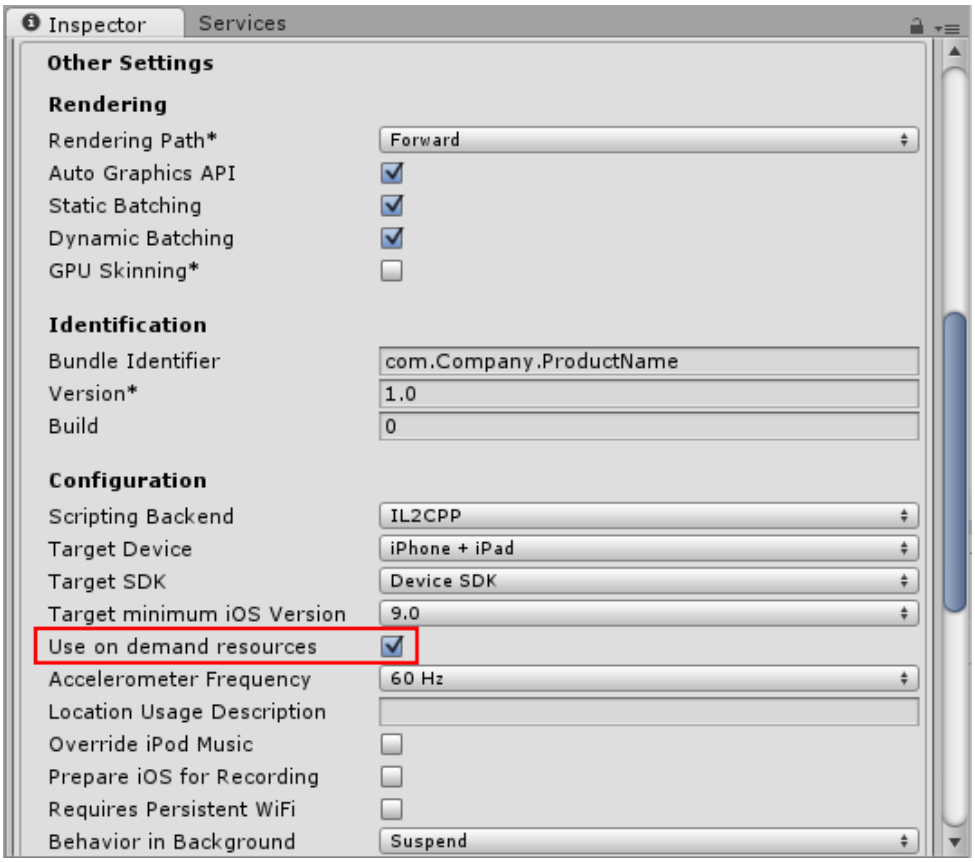
AssetBundles can contain Asset files such as models, Materials, Textures and **Scenes**, but they cannot include **scripts**. This means that all your scripting logic must reside in the main application. Apple recommends that an AssetBundle be no larger than 64MB in size, to reduce loading time (particularly over-the-air) and minimize the amount of storage space used on the device.

Setting up ODR

To set up your project for ODR, first check that your build type is set to iOS. In the menu bar, go to **File** > **Build Settings**.

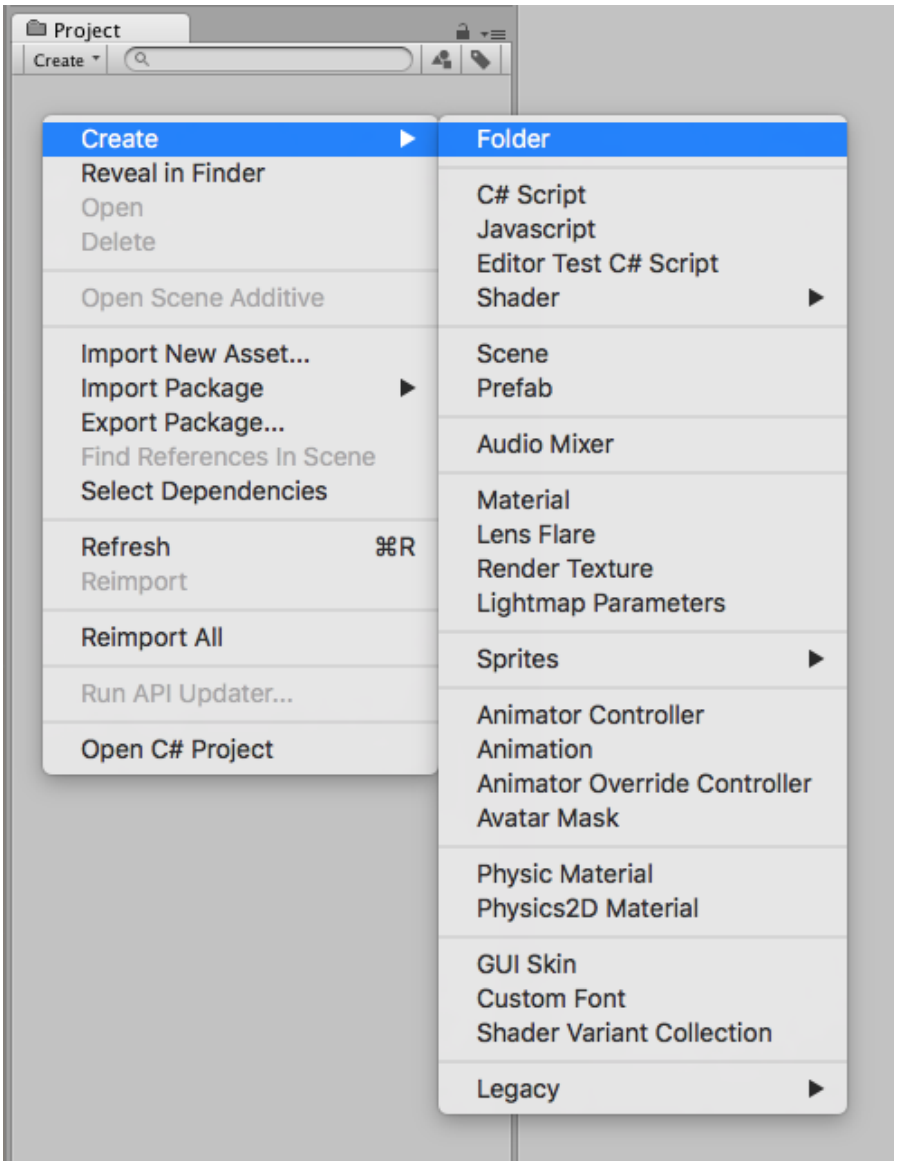


If it isn't already selected, click on **iOS**, then click the **Switch Platform** button. Next, click the **Player Settings** button. In the **Inspector** window, open **Other Settings,** navigate to the **Configuration** section, and enable the **Use on demand resources** checkbox.



Creating an AssetBundle

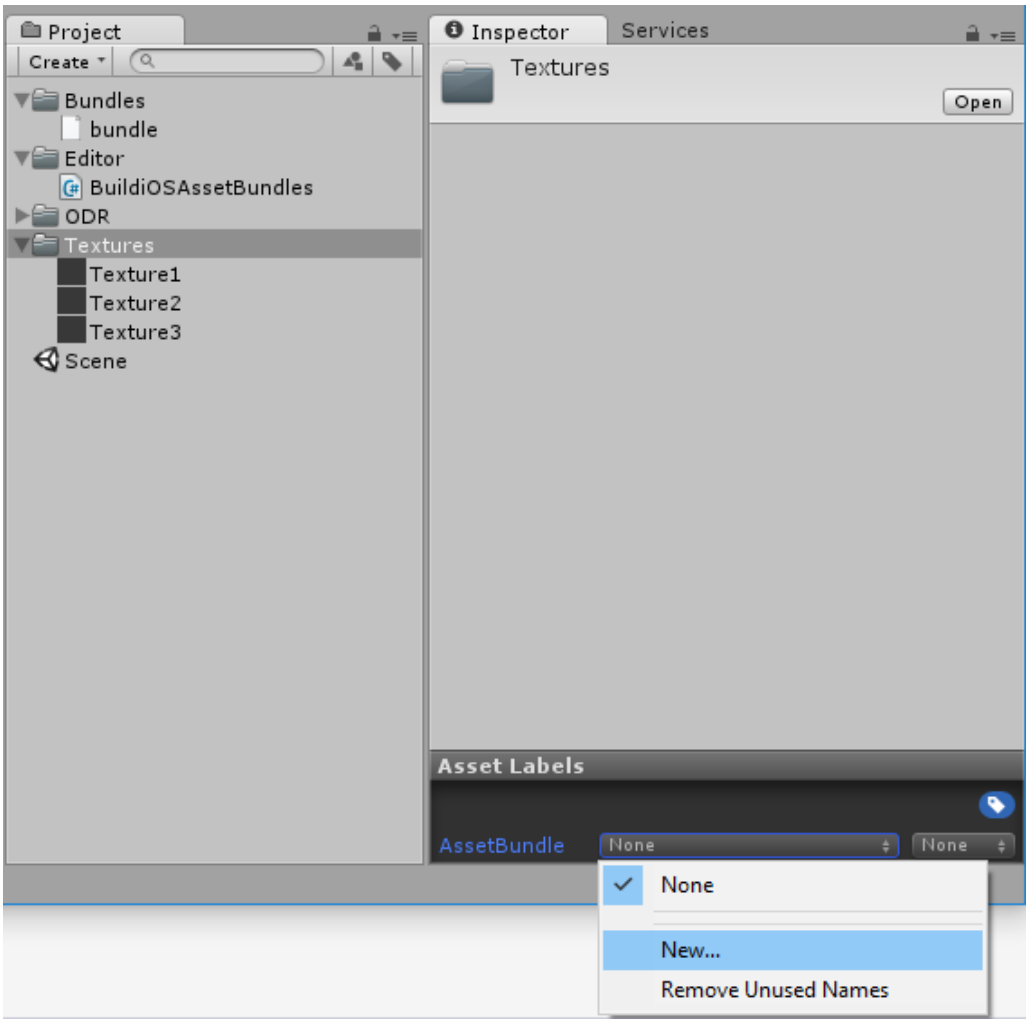
First, create a new folder to group the Assets you want to place into an AssetBundle. To do this, right-click inside the Project window and go to **Create** > **Folder** (or click **Create** > **Folder** in the top-left of the Project window).



Select the Asset files you wish to add to the AssetBundle, then drag-and-drop them into your new folder.

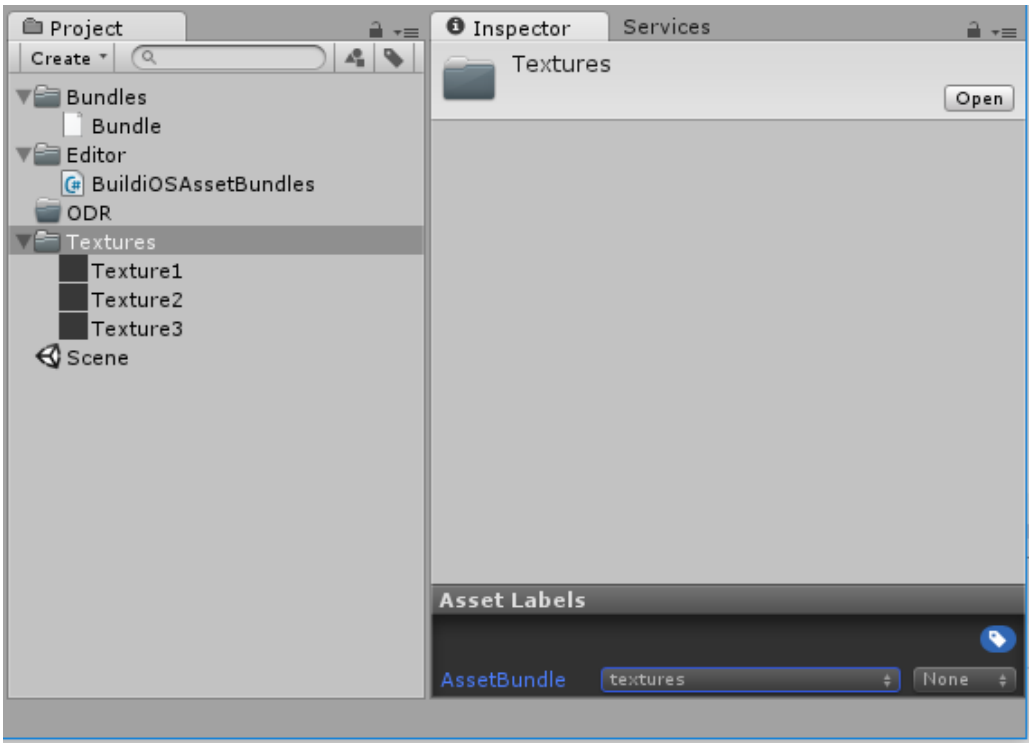
When creating an AssetBundle, you need to assign it a tag, which is used as an identifier when requesting the AssetBundle download. It's good practice to create a label that matches the bundle's filename; this ensure a unique label, and makes it easier to recognise when you are working with it later.

To create or assign a tag, select your new folder and navigate to the Asset Labels section at the bottom of the Inspector window. Click the left-hand drop-down menu, select **New**, and enter the name of your new label. Note that AssetBundle labels must be lower-case.



In this example, a folder called *Textures* is being given a new label

To generate the new AssetBundle file, the new label must be referenced in an Editor script. To create an Editor script, create a new folder inside the Project window called *Editor*. Right-click on the *Editor* folder and select **Create** > **C# Script**. Name the new script **BuildiOSAssetBundles.cs**.



In this example, the *Textures* folder has been given the label *textures*. The new script **BuildiOSAssetBundles.cs** has been created in the *Editor* folder.

Open **BuildiOSAssetBundles.cs** and copy in the code below. In this example, the label *textures* has been used; change this throughout with the name of your label (in lower-case text).

Note that this example uses uncompressed bundles; however, this is not a requirement for app thinning.

```
using UnityEngine;
using UnityEditor;


public class BuildiOSAssetBundles : MonoBehaviour
{
    [InitializeOnLoadMethod]
    static void SetupResourcesBuild( )
    {
        UnityEditor.iOS.BuildPipeline.collectResources += CollectResources;
    }

    static UnityEditor.iOS.Resource[] CollectResources( )
    {
        return new UnityEditor.iOS.Resource[]
        {
            new UnityEditor.iOS.Resource( "textures", "Assets/ODR/textures" ).AddOnDemand
            new UnityEditor.iOS.Resource( "bundle", "Assets/Bundles/bundle.unity3d" ).Add
        };
    }

    [MenuItem( "Bundle/Build iOS AssetBundle" )]
    static void BuildAssetBundles( )
```

```
    {
        var options = BuildAssetBundleOptions.None;

        bool shouldCheckODR = EditorUserBuildSettings.activeBuildTarget == BuildTarget.iO

#if UNITY_TVOS
            shouldCheckODR |= EditorUserBuildSettings.activeBuildTarget == BuildTarget.tv
#endif

        if( shouldCheckODR )
        {
#if ENABLE_IOS_ON_DEMAND_RESOURCES
            if( PlayerSettings.iOS.useOnDemandResources )
                options |= BuildAssetBundleOptions.UncompressedAssetBundle;
#endif

#if ENABLE_IOS_APP_SLICING
            options |= BuildAssetBundleOptions.UncompressedAssetBundle;
#endif
        }

        BuildPipeline.BuildAssetBundles( "Assets/ODR", options, EditorUserBuildSettings.a
    }

}
```

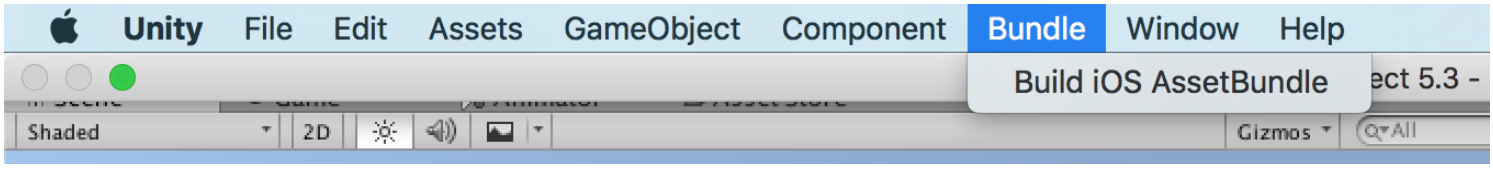
The significant line in the above code sample is the following, which takes the files tagged with the *textures* label and creates an AssetBundle file called *textures* within the Assets/ODR folder:

```
new UnityEditor.iOS.Resource( "textures", "Assets/ODR/textures" ).AddOnDemandResourceTags
```

For demonstration purposes, the above code sample also includes the following line, which adds an AssetBundle called *bundle* that has already been built - for example, from another project or a third-party vendor:

```
new UnityEditor.iOS.Resource( "bundle", "Assets/Bundles/bundle.unity3d" ).AddOnDemandReso
```

The whole code sample creates a new menu in the Unity Editor menu bar. Go to **Bundle** > **Build iOS AssetBundle**. This generates the AssetBundles in the *ODR* folder.

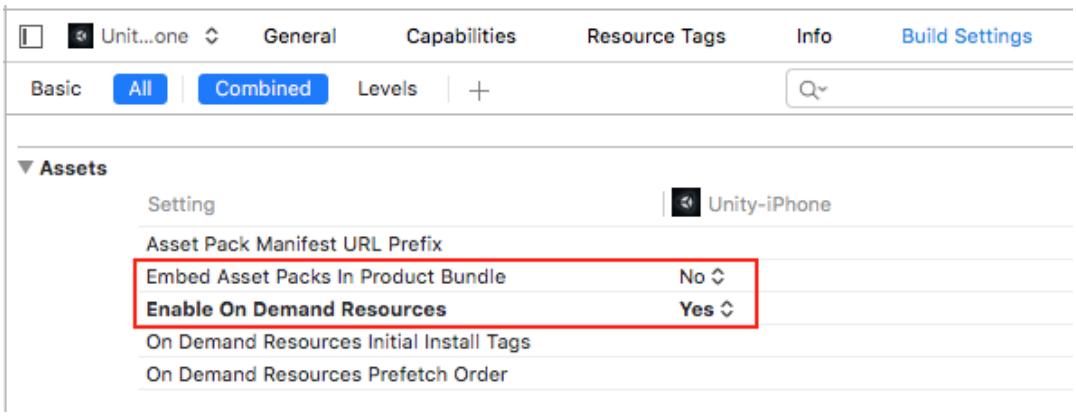


The following script downloads the *textures* ODR Asset Bundle, assigning it to the public member **TextureBundle**. Place this somewhere in your project.

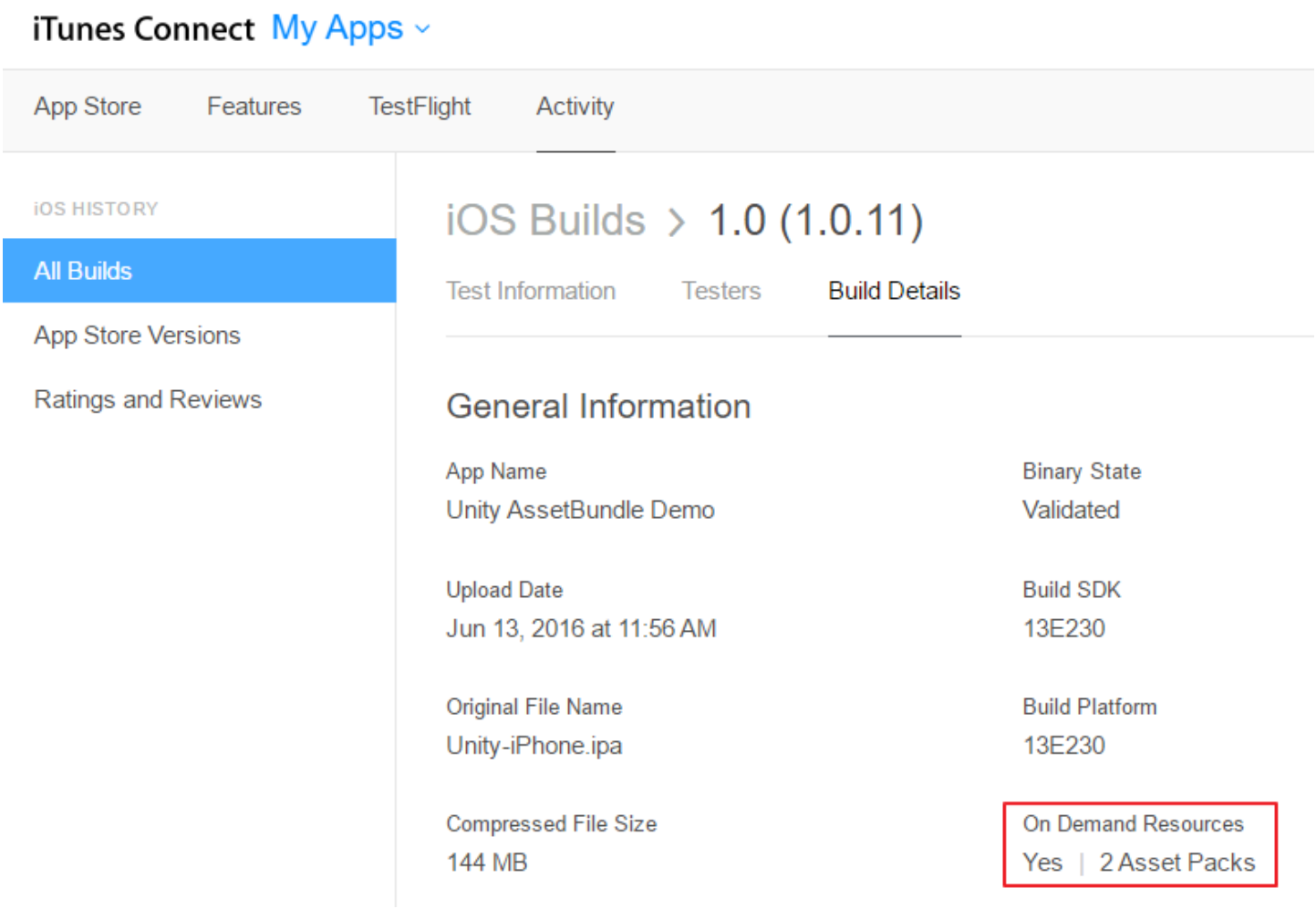
```
using UnityEngine;
using UnityEngine.iOS;
using System;
using System.Collections;

public class LoadBundle : MonoBehaviour
{
    public AssetBundle     TextureBundle;


    void Start( )
    {
        StartCoroutine( LoadAsset( "textures", "textures" ) );
    }

    public IEnumerator LoadAsset( string resourceName, string odrTag )
    {
        // Create the request
        OnDemandResourcesRequest request = OnDemandResources.PreloadAsync( new string[] {

        // Wait until request is completed
        yield return request;

        // Check for errors
        if( request.error != null )
            throw new Exception( "ODR request failed: " + request.error );

        TextureBundle = AssetBundle.LoadFromFile( "res://" + resourceName );

        request.Dispose( );
    }
}
```

The next step is to generate an Xcode project, build an .IPA, and upload it to iTunes Connect's TestFlight. As part of the TestFlight processing procedure, the embedded ODR AssetBundles are removed from the app and hosted on Apple's servers, ready for downloading.

Before you build the .IPA in Xcode, check in XCode's Build Settings to make sure that in the Assets section, **Embed Asset packs In Product Bundle** is set to **No**, and **Enable On Demand Resources** is set to **Yes**.



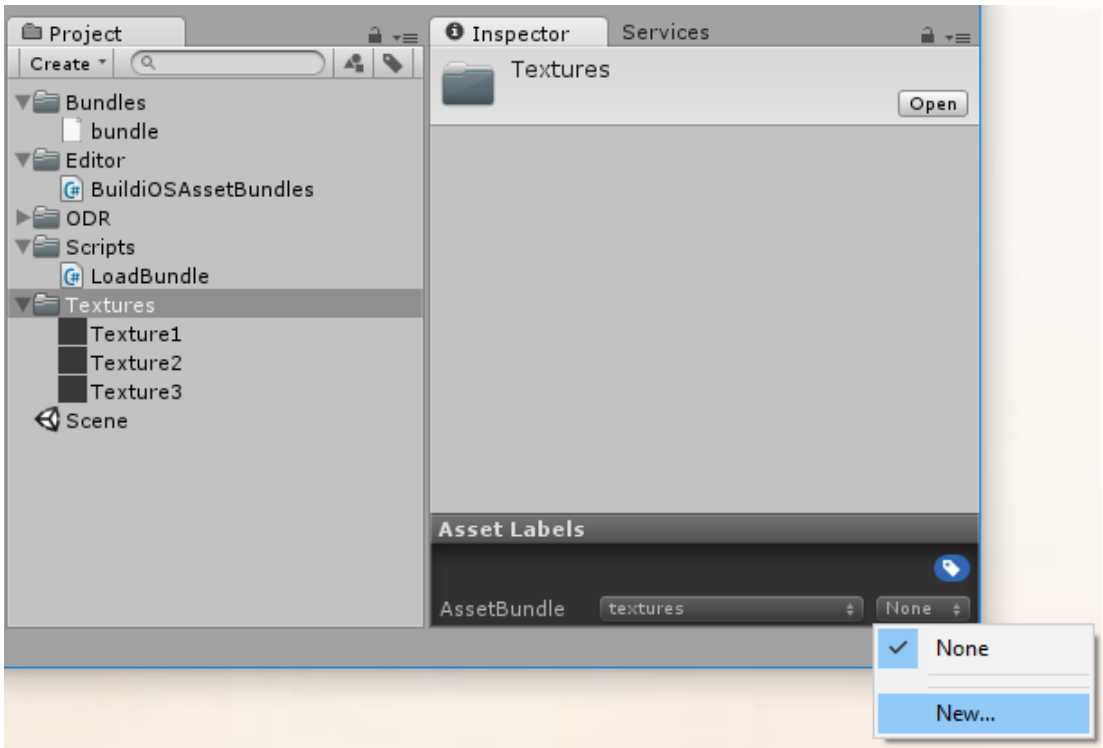
Once iTunes Connect has finished processing your app upload, click on the build version in TestFlight Builds to see more information about it:



App slicing

App slicing follows a similar process to on-demand resourcing, allowing you to dynamically download Assets according the specification of the device the app is running on (for example, to download high-resolution Assets for retina iPads, and low-resolution Assets for smaller devices like iPhones and the iPad Mini). This is achieved by defining AssetBundles, with the added provision of **variants**. This way, you can decide at startup which variant to use, and automatically append this to the Asset file name upon download.

To create a Variant, click on your new folder and navigate to the **Asset Labels** section at the bottom of the Inspector window. Click on the right-hand drop-down menu, select **New**, and enter the name of your new variant. Note that AssetBundle variants must be lower-case.



The new variant must be referenced in an Editor script. To create an Editor script, create a new folder inside the Project window called *Editor*. Right-click on the Editor folder and select **Create** > **C# Script**. Name the new script **BuildiOSAppSlices.cs**.

Copy and paste the code below, replacing the example label ("textures") and variants ("hd" and "sd") with your own. In this code example, multiple folders are referred to: one containing HD textures, and one containing SD textures. These have been given the variants "hd" and "sd" respectively.

```
using UnityEngine;
using UnityEditor;


public class BuildiOSAppSlices : MonoBehaviour
{
    [InitializeOnLoadMethod]
    static void SetupResourcesBuild( )
    {
        UnityEditor.iOS.BuildPipeline.collectResources += CollectResources;
    }

    static UnityEditor.iOS.Resource[] CollectResources( )
    {
        return new UnityEditor.iOS.Resource[]
        {
            new UnityEditor.iOS.Resource("textures").BindVariant( "Assets/ODR/textures.hd
                                        .BindVariant( "Assets/ODR/textures.sd", "sd" )
                    .AddOnDemandResourceTags( "textures" ),
    };
    }

    [MenuItem( "Bundle/Build iOS App Slices" )]
```

```
    static void BuildAssetBundles( )
    {
        var options = BuildAssetBundleOptions.None;

        bool shouldCheckODR = EditorUserBuildSettings.activeBuildTarget == BuildTarget.iO

#if UNITY_TVOS
            shouldCheckODR |= EditorUserBuildSettings.activeBuildTarget == BuildTarget.tv
#endif

        if( shouldCheckODR )
        {
#if ENABLE_IOS_ON_DEMAND_RESOURCES
            if( PlayerSettings.iOS.useOnDemandResources )
                options |= BuildAssetBundleOptions.UncompressedAssetBundle;
#endif

#if ENABLE_IOS_APP_SLICING
            options |= BuildAssetBundleOptions.UncompressedAssetBundle;
#endif
        }

        BuildPipeline.BuildAssetBundles( "Assets/ODR", options, EditorUserBuildSettings.a
    }

}
```

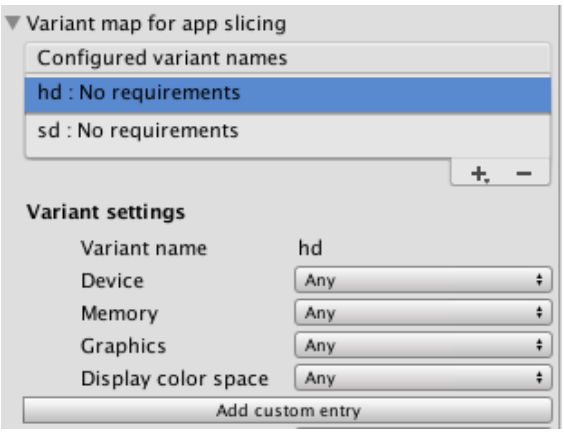
This creates a new menu in the Unity Editor menu bar called **Bundle**. Click this and select the only item in the list, **Build iOS App Slices**. This generates the AssetBundles in the **ODR** folder.

Then, to load an Asset, place this class somewhere in your project and pass in the name of the Variant you wish to load:

```
using UnityEngine;
using UnityEngine.iOS;
using System;
using System.Collections;

public class LoadBundle : MonoBehaviour
{
    public AssetBundle     TextureBundle;


    void Start( )
    {
        StartCoroutine( LoadAsset( "textures.hd", "textures" ) );
    }

    public IEnumerator LoadAsset( string resourceName, string odrTag )
```

```
    {
        // Create the request
        OnDemandResourcesRequest request = OnDemandResources.PreloadAsync( new string[] {

        // Wait until request is completed
        yield return request;

        // Check for errors
        if( request.error != null )
            throw new Exception( "ODR request failed: " + request.error );

        TextureBundle = AssetBundle.LoadFromFile( "res://" + resourceName );

        request.Dispose( );
    }
}
```

You can now see and modify variants in **Player Settings > Other Settings > Configuration** in a **Variant map for app slicing** drop-down menu (the menu is only visible if **Use on demand resources** is enabled in the Player Settings).



To learn more about AssetBundles and on-demand resources, see Unity's AssetBundle Manager demo project, hosted on Bitbucket. The landing page offers a comprehensive description of how to use and tweak the demo.

2018–06–14 Page amended with limited editorial review

Managed stack traces on iOS

Leave feedback

When an exception occurs in managed code, the stack trace for the exception can help you understand the cause of the exception. However, the managed stack trace might not appear as expected in some cases on **iOS**, as explained below. The stack trace also varies depending on the Xcode build configuration.

Debug builds

When you use the debug build configuration with iOS, **IL2CPP** should report a reliable managed stack trace, and include each managed method in the call stack. The stack trace does not include line numbers from the original C# source code.

Release builds

When you use a release build configuration, IL2CPP might produce a call stack that's missing one or more managed methods. This is because the C++ compiler has inlined the missing methods. Method inlining is usually good for performance at run time, but it can make call stacks more difficult to understand. IL2CPP always provides at least one managed method on the call stack. This is the method where the exception occurred. It also includes other methods if they are not inlined.

If you can reproduce an exception locally, use Xcode to determine which methods have been inlined. Run the application in Xcode using the release configuration and set an exception breakpoint. The native call stack view in Xcode should indicate which methods actually exist, and which have been inlined.

Source code line numbers

IL2CPP call stacks do not include source code line number information in the debug or release configurations.

---

21 March 2018 Page published with editorial review

Features currently not supported by Unity iOS

Leave feedback

Graphics

DXT texture **compression** is not supported; use other formats instead. Please see the Texture2D Component page for more information.
Rectangular textures cannot be compressed to **PVRTC** formats.
Movie Textures are not supported; use a full-screen streaming playback instead. Please see the Movie playback page for more information.

Scripting

Dynamic features like Duck Typing are not supported. Use `#pragma strict` for your **scripts** to force the compiler to report dynamic features as errors.
Video streaming via `WWW` and `WebRequest` classes is not supported.
FTP support by `WWW` and `WebRequest` classes is limited.

Features restricted to Unity iOS Pro License

Splash-screen customization

External libraries

It is recommended to minimize your references to external libraries, because 1 MB of .NET CIL code roughly translates to 3–4 MB of ARM code. For example, if your application references *System.dll* and *System.Xml.dll* then it means additional 6 MB of ARM code **if stripping is not used**. At some point the application will reach this limit and the linker will have trouble linking the code.

2018–06–14 Page amended with limited editorial review

Troubleshooting on iOS devices

Leave feedback

There are some situations with **iOS** where your game can work perfectly in the Unity editor but then doesn't work or maybe doesn't even start on the actual device. The problems are often related to code or content quality. This section describes the most common scenarios.

The game stops responding after a while. Xcode shows "interrupted" in the status bar.

There are a number of reasons why this may happen. Typical causes include:

Scripting errors such as using uninitialized variables, **etc**.
Using 3rd party Thumb compiled native libraries. Such libraries trigger a known problem in the iOS SDK linker and might cause random crashes.
Using generic types with value types as parameters (e.g. List<int>, List<SomeStruct>, List<SomeEnum>) for serializable script properties.
Using reflection when managed code stripping is enabled.
Errors in the **native plugin** interface (the managed code method signature does not match the native code function signature). Information from the XCode Debugger console can often help detect these problems (Xcode menu: **View > Debug Area > Activate Console**).

The Xcode console shows "Program received signal: "SIGBUS" or EXC_BAD_ACCESS error.

This message typically appears on iOS devices when your application receives a NullReferenceException. There are two ways to figure out where the fault happened:

Managed stack traces

Unity includes software-based handling of the NullReferenceException. The AOT compiler includes quick checks for null references each time a method or variable is accessed on an object. This feature affects script performance, which is why it is enabled only for **development builds** (enable the *script debugging* option in build settings dialog). If everything was done right and the fault actually is occurring in .NET code then you won't see EXC_BAD_ACCESS anymore. Instead, the .NET exception text will be printed in the Xcode console (or else your code will just handle it in a "catch" statement). Typical output might be:

```
Unhandled Exception: System.NullReferenceException: A null value was found where an object instai
  at DayController+$handleTimeOfDay$121+$.MoveNext () [0x0035a] in DayController.js:122
```

This indicates that the fault happened in the handleTimeOfDay method of the DayController class, which works as a coroutine. Also, if it is script code then you will generally be told the exact line number (e.g. "DayController.js:122"). The offending line might be something like the following:

```
Instantiate(_imgwww.assetBundle.mainAsset);
```

This might happen if, say, the script accesses an **asset** bundle without first checking that it was downloaded correctly.

Native stack traces

Native stack traces are a much more powerful tool for fault investigation but using them requires some expertise. Also, you generally can't continue after these native (hardware memory access) faults happen. To get a native stack trace, type **bt all** into the Xcode Debugger Console. Carefully inspect the printed stack traces; they may contain hints about where the error occurred. You might see something like:

```
...
Thread 1 (thread 11523):

1. 0 0x006267d0 in m_OptionsMenu_Start ()
1. 1 0x002e4160 in wrapper_runtime_invoke_object_runtime_invoke_void__this___object_intptr_intpt
1. 2 0x00a1dd64 in mono_jit_runtime_invoke (method=0x18b63bc, obj=0x5d10cb0, params=0x0, exc=0x2
1. 3 0x0088481c in MonoBehaviour::InvokeMethodOrCoroutineChecked ()
...
```

Firstly, you should find the stack trace for **"Thread 1"**, which is the main thread. The very first lines of the stack trace will point to the place where the error occurred. In this example, the trace indicates that the NullReferenceException happened inside the *OptionsMenu* script's *Start* method. Looking carefully at this method implementation would reveal the cause of the problem. Typically, NullReferenceExceptions happen inside the *Start* method when incorrect assumptions are made about initialization order. In some cases only a partial stack trace is seen on the Debugger Console:

```
Thread 1 (thread 11523):

1. 0 0x0062564c in start ()
```

This indicates that native symbols were stripped during the Release build of the application. The full stack trace can be obtained with the following procedure:

Remove application from device.
Clean all targets.
Build and run.
Get stack traces again as described above.

EXC_BAD_ACCESS starts occurring when an external library is linked to the Unity iOS application.

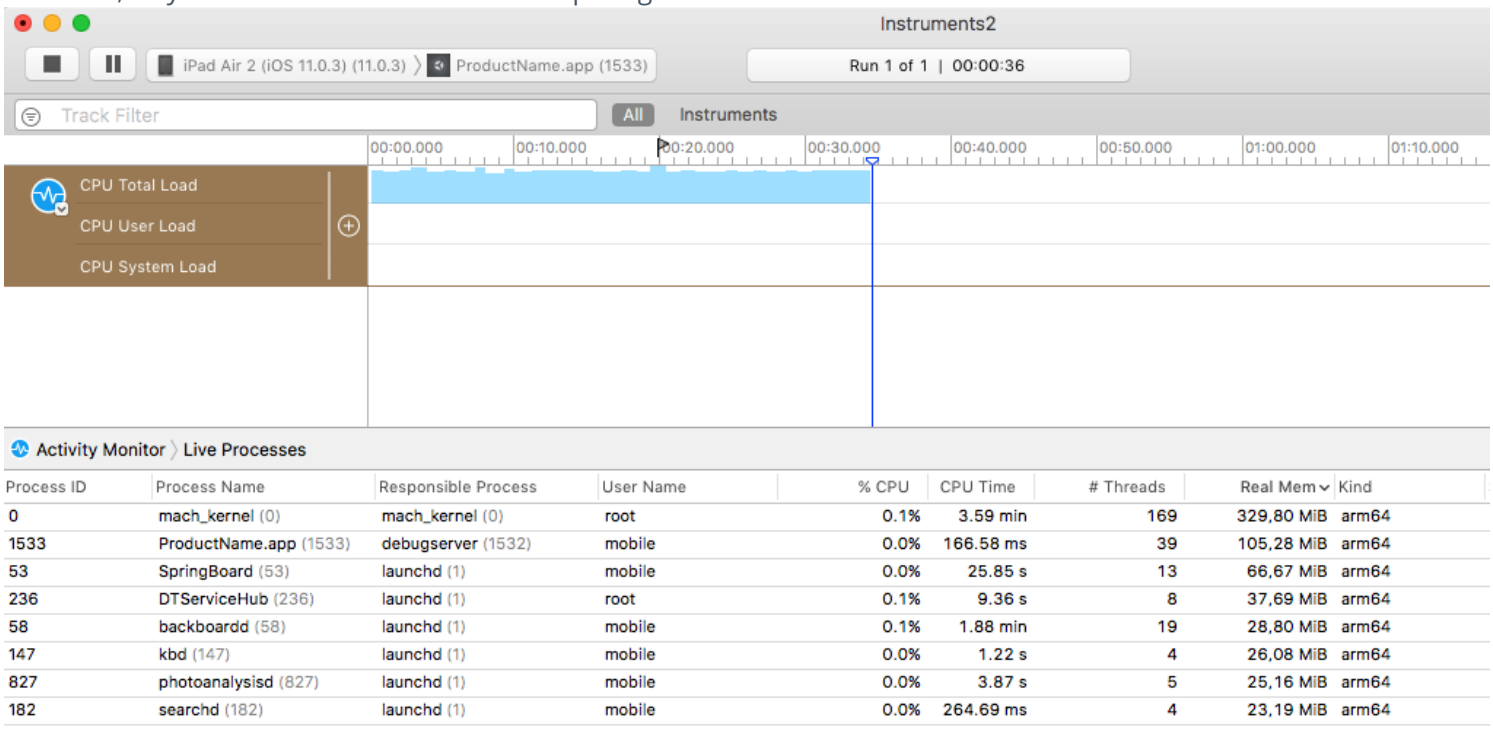
This usually happens when an external library is compiled with the ARM Thumb instruction set. Currently, such libraries are not compatible with Unity. The problem can be solved easily by recompiling the library without Thumb instructions. You can do this for the library's Xcode project with the following steps:

In Xcode, select *View* > *Navigators* > *Show Project Navigator* from the menu
Select the *Unity-iPhone* project, activate *Build Settings* tab
In the search field enter: *Other C Flags*
Add *-mno-thumb* flag there and rebuild the library.
If the library source is not available you should ask the supplier for a non-thumb version of the library.

The Xcode console shows "WARNING -> applicationDidReceiveMemoryWarning()" and the application crashes immediately afterwards

Sometimes, you might see a message like *Program received signal: "0"*. This warning message is often not fatal and merely indicates that iOS is low on memory and is asking applications to free up some memory. Typically, background processes like Mail will free some memory and your application can continue to run. However, if your application continues to use memory or ask for more, the OS will eventually start killing applications and yours could be one of them. Apple does not document what memory usage is safe, but empirical observations show that applications using less than 50% of all device RAM do not have major memory usage problems. The main metric you should rely on is how much RAM your application uses. Your application memory usage consists of three major **components**:

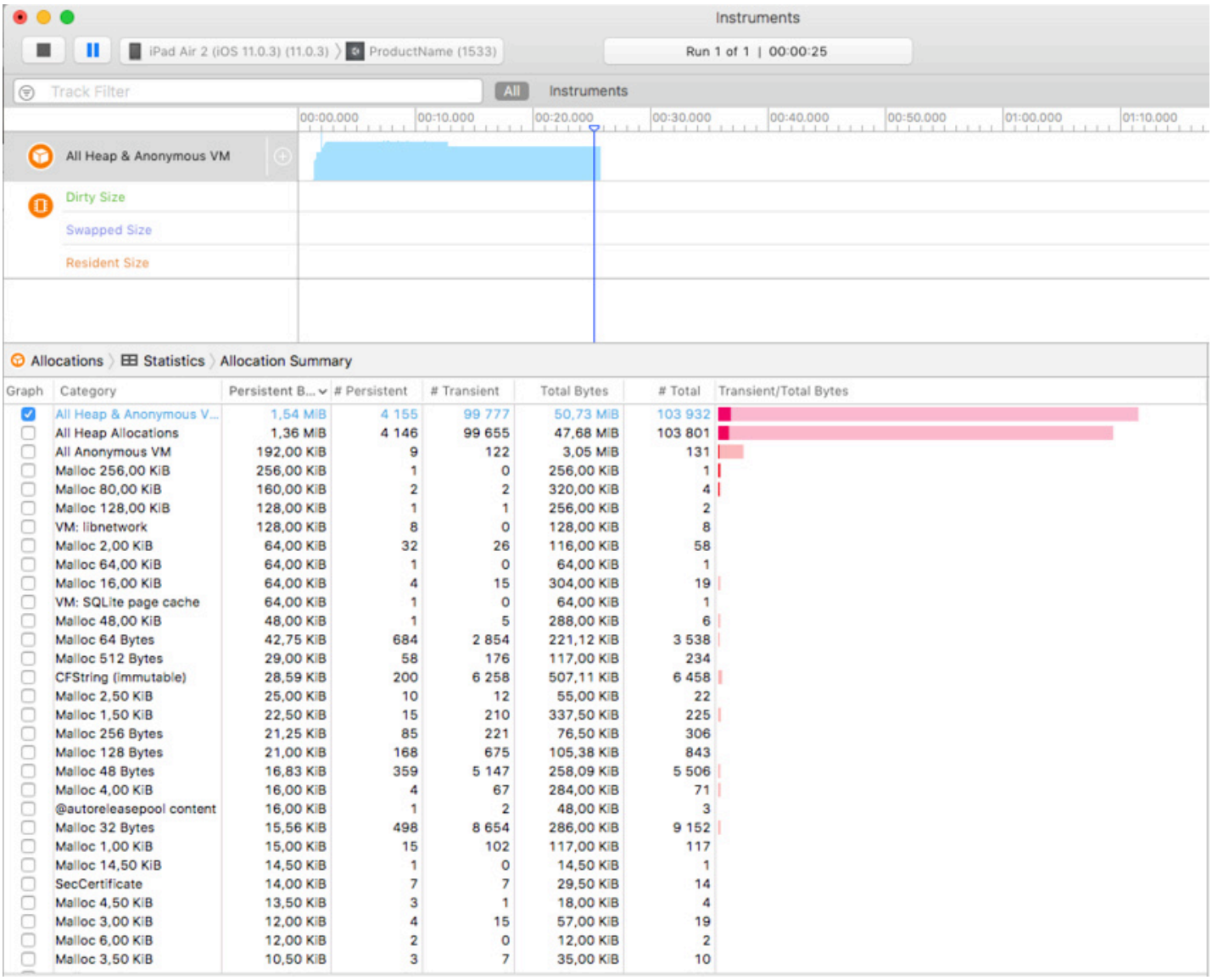
Application code (the OS needs to load and keep your application code in RAM, but some of it might be discarded if really needed)
Native heap (used by the engine to store its state, your assets, etc. in RAM)
Managed heap (used by your Mono runtime to keep C# objects)
GLES driver memory pools: textures, framebuffers, compiled **shaders**, etc. Your application memory usage can be tracked by two Xcode Instruments tools: **Activity Monitor** and **Allocations**. You can start from the Xcode Run menu: **Product > Profile** and then select specific tool or via **Xcode > Open Developer Tools > Instruments**. **Activity Monitor** tool shows all process statistics including **Real memory** which can be regarded as the total amount of RAM used by your application. **Note:** OS and device HW version combination might noticeably affect memory usage numbers, so you should be careful when comparing numbers obtained on different devices.



Process ID	Process Name	Responsible Process	User Name	% CPU	CPU Time	# Threads	Real Mem	Kind
0	mach_kernel (0)	mach_kernel (0)	root	0.1%	3.59 min	169	329,80 MiB	arm64
1533	ProductName.app (1533)	debugserver (1532)	mobile	0.0%	166.58 ms	39	105,28 MiB	arm64
53	SpringBoard (53)	launchd (1)	mobile	0.0%	25.85 s	13	66,67 MiB	arm64
236	DTServiceHub (236)	launchd (1)	root	0.1%	9.36 s	8	37,69 MiB	arm64
58	backboardd (58)	launchd (1)	mobile	0.1%	1.88 min	19	28,80 MiB	arm64
147	kbd (147)	launchd (1)	mobile	0.0%	1.22 s	4	26,08 MiB	arm64
827	photoanalysisd (827)	launchd (1)	mobile	0.0%	3.87 s	5	25,16 MiB	arm64
182	searchd (182)	launchd (1)	mobile	0.0%	264.69 ms	4	23,19 MiB	arm64

Note: The internal profiler shows only the heap allocated by .NET **scripts**. Total memory usage can be determined via Xcode Instruments as shown above. This figure includes parts of the application binary, some standard framework buffers, Unity engine internal state buffers, the .NET runtime heap (number printed by internal profiler), GLES driver heap and some other miscellaneous stuff.

The other tool displays all allocations made by your application and includes both native heap and managed heap statistics. The important statistic is the **Net bytes** value.



Graph	Category	Persistent B...	# Persistent	# Transient	Total Bytes	# Total	Transient/Total Bytes
☑	All Heap & Anonymous V...	1,54 MiB	4 155	99 777	50,73 MiB	103 932	
☐	All Heap Allocations	1,36 MiB	4 146	99 655	47,68 MiB	103 801	
☐	All Anonymous VM	192,00 KiB	9	122	3,05 MiB	131	
☐	Malloc 256,00 KiB	256,00 KiB	1	0	256,00 KiB	1	
☐	Malloc 80,00 KiB	160,00 KiB	2	2	320,00 KiB	4	
☐	Malloc 128,00 KiB	128,00 KiB	1	1	256,00 KiB	2	
☐	VM: libnetwork	128,00 KiB	8	0	128,00 KiB	8	
☐	Malloc 2,00 KiB	64,00 KiB	32	26	116,00 KiB	58	
☐	Malloc 64,00 KiB	64,00 KiB	1	0	64,00 KiB	1	
☐	Malloc 16,00 KiB	64,00 KiB	4	15	304,00 KiB	19	
☐	VM: SQLite page cache	64,00 KiB	1	0	64,00 KiB	1	
☐	Malloc 48,00 KiB	48,00 KiB	1	5	288,00 KiB	6	
☐	Malloc 64 Bytes	42,75 KiB	684	2 854	221,12 KiB	3 538	
☐	Malloc 512 Bytes	29,00 KiB	58	176	117,00 KiB	234	
☐	CFString (immutable)	28,59 KiB	200	6 258	507,11 KiB	6 458	
☐	Malloc 2,50 KiB	25,00 KiB	10	12	55,00 KiB	22	
☐	Malloc 1,50 KiB	22,50 KiB	15	210	337,50 KiB	225	
☐	Malloc 256 Bytes	21,25 KiB	85	221	76,50 KiB	306	
☐	Malloc 128 Bytes	21,00 KiB	168	675	105,38 KiB	843	
☐	Malloc 48 Bytes	16,83 KiB	359	5 147	258,09 KiB	5 506	
☐	Malloc 4,00 KiB	16,00 KiB	4	67	284,00 KiB	71	
☐	@autoreleasepool content	16,00 KiB	1	2	48,00 KiB	3	
☐	Malloc 32 Bytes	15,56 KiB	498	8 654	286,00 KiB	9 152	
☐	Malloc 1,00 KiB	15,00 KiB	15	102	117,00 KiB	117	
☐	Malloc 14,50 KiB	14,50 KiB	1	0	14,50 KiB	1	
☐	SecCertificate	14,00 KiB	7	7	29,50 KiB	14	
☐	Malloc 4,50 KiB	13,50 KiB	3	1	18,00 KiB	4	
☐	Malloc 3,00 KiB	12,00 KiB	4	15	57,00 KiB	19	
☐	Malloc 6,00 KiB	12,00 KiB	2	0	12,00 KiB	2	
☐	Malloc 3,50 KiB	10,50 KiB	3	7	35,00 KiB	10	

To keep memory usage low:

Reduce the application binary size by using the strongest iOS stripping options, and avoid unnecessary dependencies on different .NET libraries. See the player settings and player size optimization manual pages for further details.
Reduce the size of your content. Use **PVRTC compression** for textures and use low poly models. See the manual page about reducing file size for more information.
Don't allocate more memory than necessary in your scripts. Track mono heap size and usage with the internal profiler.
Querying the OS about the amount of free memory may seem like a good idea to evaluate how well your application is performing. However, the free memory statistic is likely to be unreliable since the OS uses a lot of dynamic buffers and caches. The only reliable approach is to keep track of memory consumption for your application and use that as the main metric. Pay attention to how the graphs from the tools described above change over time, especially after loading new levels.

The game runs correctly when launched from Xcode but crashes while loading the first level when launched manually on the device.

There could be several reasons for this. You need to inspect the device logs to get more details. Connect the device to your Mac, launch Xcode and select **Window > Devices and Simulators** from the menu. Select your device in the window's left toolbar, then click on the **Show the device console** button and review the latest messages carefully. Additionally, you may need to investigate crash reports. You can find out how to obtain crash reports here: http://developer.apple.com/iphone/library/technotes/tn2008/tn2151.html.

The Xcode Organizer console contains the message "killed by SpringBoard".

There is a poorly-documented time limit for an iOS application to render its first frames and process input. If your application exceeds this limit, it will be killed by SpringBoard. This may happen in an application with a first **scene** which is too large, for example. To avoid this problem, it is advisable to create a small initial **scene** which just displays a splash screen, waits a frame or two with **yield** and then starts loading the real **scene**. This can be done with code as simple as the following:

```
IEnumerator Start() {
    yield return new WaitForEndOfFrame();
// Do not forget using UnityEngine.SceneManagement directive
    SceneManager.LoadScene("Test");
}
```

Type.GetProperty() or Type.GetValue() cause crashes on the device

Currently **Type.GetProperty()** and **Type.GetValue()** are supported only for the **.NET 2.0 Subset** profile. You can select the .NET API compatibility level in the Player Settings.

Note: Type.GetProperty() and **Type.GetValue()** might be incompatible with managed code stripping and might need to be excluded (you can supply a custom non-strippable type list during the stripping process to accomplish this). For further details, see the iOS player size optimization guide.

The game crashes with the error message "ExecutionEngineException: Attempting to JIT compile method 'SometType`1<SomeValueType>:.ctor ()' while running with –aot-only."

The Mono .NET implementation for iOS is based on AOT (ahead of time compilation to native code) technology, which has its limitations. It compiles only those generic type methods (where a value type is used as a generic parameter) which are explicitly used by other code. When such methods are used only via reflection or from native code (i.e. the serialization system) then they get skipped during **AOT compilation**. The AOT compiler can be hinted to include code by adding a dummy method somewhere in the script code. This can refer to the missing methods and so get them compiled ahead of time.

```
void _unusedMethod() {
    var tmp = new SomeType<SomeValueType>();
}
```

Note: Value types are basic types, enums and structs.

Various crashes occur on the device when a combination of System.Security.Cryptography and managed code stripping is used

.NET Cryptography services rely heavily on reflection and so are not compatible with managed code stripping since this involves static code analysis. Sometimes the easiest solution to the crashes is to exclude the whole **System.Security.Crypography** namespace from the stripping process.

The stripping process can be customized by adding a custom **link.xml** file to the *Assets* folder of your Unity project. This specifies which types and namespaces should be excluded from stripping. Further details can be found in the iOS player size optimization guide.

link.xml

```
<linker>
       <assembly fullname="mscorlib">
```

```
                <namespace fullname="System.Security.Cryptography" preserve="all"/>
        </assembly>
</linker>
```

Application crashes when using System.Security.Cryptography.MD5 with managed code stripping

You should consider the above advice or try working around this problem by adding extra references to specific classes to your script code:

```
object obj = new MD5CryptoServiceProvider();
```

“Ran out of trampolines of type 0/1/2” runtime error

This error usually happens if you use lots of recursive generics. You can hint to the AOT compiler to allocate more trampolines of type 0, type 1 or type 2. Additional AOT compiler command line options can be specified in the *Other Settings* section of the Player Settings. For type 1 trampolines, specify `nrgctx-trampolines=ABCD`, where ABCD is the number of new trampolines required (i.e. 4096). For type 2 trampolines specify `nimt-trampolines=ABCD` and for type 0 trampolines specify `ntrampolines=ABCD`.

After upgrading Xcode Unity iOS, runtime fails with message “You are using Unity iPhone Basic. You are not allowed to remove the Unity splash screen from your game”

With some latest Xcode releases there were changes introduced in PNG compression and optimization tool. These changes might cause false positives in Unity iOS runtime checks for splash screen modifications. If you encounter such problems try upgrading Unity to the latest publicly available version. If this does not help, you might consider the following workaround:

Replace your Xcode project from scratch when building from Unity (instead of appending it)
Delete already installed projects from device
Clean project in Xcode (*Product* > *Clean*)
Clear Xcode’s Derived Data folders (*Xcode* > *Preferences* > *Locations*)
If this still does not help try disabling PNG re-compression in Xcode:

Open your Xcode project
Select *Unity-iPhone* project
Select **Build Settings** tab
Look for **Compress PNG files** option and set to **NO**

WWW downloads are working fine in Unity Editor and on Android, but not on iOS

The most common mistake is to assume that WWW downloads are always happening on a separate thread. On some platforms this might be true, but you should not take it for granted. Best way to track WWW status is either to use the *yield* statement or check status in *Update* method. You should **not** use busy *while* loops for that.

“PlayerLoop called recursively!” error occurs when using Cocoa via a native function called from a script

Some operations with the **UI** will result in iOS redrawing the window immediately (the most common example is adding a UIView with a UIViewController to the main UIWindow). If you call a native function from a script, it will happen inside Unity’s PlayerLoop, resulting in PlayerLoop being called recursively. In such cases, you should consider using the performSelectorOnMainThread method with *waitUntilDone* set to false. It will inform iOS to schedule the operation to run between Unity’s PlayerLoop calls.

Profiler or Debugger unable to see game running on iOS device

Check that you have built a Development build, and checked the *Script Debugging* and *Autoconnect Profiler* boxes (as appropriate).
The application running on the device will make a multicast broadcast to 225.0.0.222 on UDP port 54997. Check that your network settings allow this traffic. Then, the **profiler** will make a connection to the remote device on a port in the range 55000 - 55511 to fetch **profiler** data from the device. These ports will need to be open for UDP access.

Missing DLLs

If your application runs ok in editor but you get errors in your iOS project this may be caused by missing DLLs (e.g. I18N.dll, I19N.West.dll). In this case, try copying those dlls from within the Unity.app to your project's *Assets\Plugins* folder. The location of the DLLs within the unity app is: *Unity.app\Contents\Frameworks\Mono\lib\mono\unity* You should then also check the stripping level of your project to ensure the classes in the DLLs aren't being removed when the build is optimised. Refer to the iOS Optimisation Page for more information on iOS Stripping Levels.

Xcode Debugger console reports: ExecutionEngineException: Attempting to JIT compile method '(wrapper native-to-managed) Test:TestFunc (int)' while running with –aot-only

Typically, such a message is received when the managed function delegate is passed to the native function, but the required wrapper code wasn't generated when the application was built. You can help AOT compiler by hinting which methods will be passed as delegates to the native code. This can be done by adding the *MonoPInvokeCallbackAttribute* custom attribute. Currently, only static methods can be passed as delegates to the native code.

Sample code:

```
using UnityEngine;
using System.Collections;
using System;
using System.Runtime.InteropServices;
using AOT;

public class NewBehaviourScript : MonoBehaviour {
    [DllImport ("__Internal")]
    private static extern void DoSomething (NoParamDelegate del1, StringParamDelegate del2);

    delegate void NoParamDelegate ();
    delegate void StringParamDelegate (string str);

    [MonoPInvokeCallback(typeof(NoParamDelegate))]
    public static void NoParamCallback() {
        Debug.Log ("Hello from NoParamCallback");
    }

    [MonoPInvokeCallback(typeof(StringParamDelegate))]
    public static void StringParamCallback(string str) {
        Debug.Log(string.Format("Hello from StringParamCallback {0}", str));
    }

    // Use this for initialization
    void Start() {
        DoSomething(NoParamCallback, StringParamCallback);
    }
}
```

Xcode throws compilation error: “ld : unable to insert branch island. No insertion point available. for architecture armv7”, “clang: error: linker command failed with exit code 1 (use -v to see invocation)”

This error usually means there is just too much code in single module. Typically, it is caused by having lots of script code or having big external .NET assemblies included into build. And enabling script debugging might make things worse, because it adds quite few additional instructions to each function, so it is easier to hit that limit.

Enabling managed code stripping in player settings might help with this problem, especially if big external .NET assemblies are involved. But if the issue persists then the best solution is to split user script code into multiple assemblies. The easiest way to this is move some code to *Plugins* folder. Code at this location is put to a different assembly. Also, check the information about how special folder names affect script compilation.

2018–06–14 Page amended with limited editorial review

Reporting crash bugs on iOS

Leave feedback

Before submitting a bug report, please check the iOS Troubleshooting page, where you will find solutions to common crashes and other problems.

If your application crashes in the Xcode debugger then you can add valuable information to your bug report as follows:-

Click Continue (**Debug** > **Continue**) twice
Open the debugger console (**View** > **Debug Area** > **Activate Console**) and enter (in the console):
thread backtrace all
Copy **all** console output and send it together with your bug report.
Similarly, when reporting app freezes, you can click **Pause** and do the above steps.

If your application crashes on the **iOS** device, you should retrieve the crash report as described here on Apple's website. Please attach the crash report, your built application and console log to your bug report before submitting.

2018–06–14 Page amended with limited editorial review

Android

Leave feedback

This section of the User Manual contains documentation on developing for the Android platform. Use the navigation column on the left-hand side of the page to view topics in this section.

Getting started with Android development

Leave feedback

The Android environment setup topic of the Unity Manual contains a basic outline of the tasks that you must complete before you are able to run code on your Android device, or in the Android emulator. For more in-depth information on setting your Android development environment, see the step-by-step instructions on the Android developer portal.

If you miss installing some necessary item during set-up, Unity verifies your development environment when building for Android and prompts you to upgrade or download missing **components**.

Unity provides scripting APIs that allow you to access various input data and settings from Android devices.

Refer to the Android scripting page of the Manual for more information.

Exposing native C, C++ or Java code to scripts

Use plug-ins to call Android functions written in C/C++ directly from C# **scripts** (Java functions can be called indirectly).

To find out how to make these functions accessible from within Unity, visit the Android plug-ins page.

Occlusion culling

Unity includes support for occlusion culling, which is a valuable optimization method for mobile platforms.

Refer to the Occlusion Culling Manual page for more information.

Splash screen customization

You can customize the splash screen that displays while the game launches on Android.

Refer to the Customizing an Android Splash Screen Manual page for more information.

Troubleshooting and bug reports

The Android troubleshooting guide helps you discover the cause of bugs as quickly as possible. If, after consulting the guide, you suspect the problem is being caused by Unity, file a bug report following the Unity bug reporting guidelines.

See the Android bug reporting page for details about filing bug reports.

Texture compression

Ericsson Texture **Compression** (ETC) is the standard **texture compression** format on Android.

ETC1 is supported on all current Android devices, but it does not support textures that have an alpha channel. ETC2 is supported on all Android devices that support OpenGL ES 3.0. It provides improved quality for RGB textures, and also supports textures with an alpha channel.

By default, Unity uses ETC1 for compressed RGB textures and ETC2 for compressed RGBA textures. If ETC2 is not supported by an Android device, the texture is decompressed at run time. This has an impact on memory usage, and also affects **rendering** speed.

DXT, **PVRTC**, **ATC**, and **ASTC** are all support textures with an alpha channel. These formats also support higher compression rates and/or better image quality, but they are only supported on a subset of Android devices.

It is possible to create separate Android distribution archives (.apk) for each of these formats and let the Android Market's filtering system select the correct archives for different devices.

Movie/Video playback

We recommend you use the Video Player to play video files. This supersedes the earlier Movie Texture feature.

Video Player component added in Unity 5.6

Android environment setup

Leave feedback

Whether you're building an Android application in Unity or programming it from scratch, you must set up the Android Software Development Kit (SDK) before you can build and run any code on your Android device.

1. Install the Java Development Kit

Download and install the Java Development Kit (JDK). Unity requires the 64-bit version JDK 8 (1.8).

2. Download the Android SDK

You can install the Android SDK using command line tools or through Android Studio. Android Studio provides an easy to use GUI based tool, but installs additional software on your computer. Using the command line tools is a smaller download and does not install additional software, but it can be more challenging to use.

2a. Install the Android SDK using the command line tools

Install or unpack the Android SDK. After installing, open the Android SDK Manager and add: at least one Android SDK Platform, the Platform Tools, the Build Tools, and the USB drivers if you're using Windows.

To install an Android platform SDK and the associated tools:

Download the Android Software command line tool.

Unzip the tools folder to a location on your hard drive.

Open a command-prompt window.

Navigate to the bin folder in the location where you unzipped the tools folder: "install folder" > tools > bin

Use the sdkmanager command line tool to retrieve the list of packages that you can install. The installable packages include the Platform SDKs, Build Tools, Platform tools, and other tools.

sdkmanager –list

Select a version of the Platform SDK to install. Platform SDKs take the following form in the list: platforms;android-xx. The *xx* indicates the SDK level. The larger the number, the newer the package. Typically, you can install the latest available version. But, there might be cases in which Google has released a new version of the SDK that causes errors when you build your Unity Project. In that case you must uninstall the SDK and install an earlier version. The general format of the command for package installation is *sdkmanager <package name>*. You can install the corresponding Platform Tools and Build Tools at the same time.

Example: *sdkmanager "platform-tools" "platforms;android–27" "build-tools;27.0.3"*

If you are running on Windows, install the USB device drivers.

sdkmanager "extras;google;usb_driver"

This installs the SDK in a directory named "platforms" in the directory in which you unzipped the tools folder. Example: c:\<install folder>\platforms

2b. Install the SDK using Android Studio

Install Android studio from the Android developer portal. The Android developer portal provides detailed installation instructions.

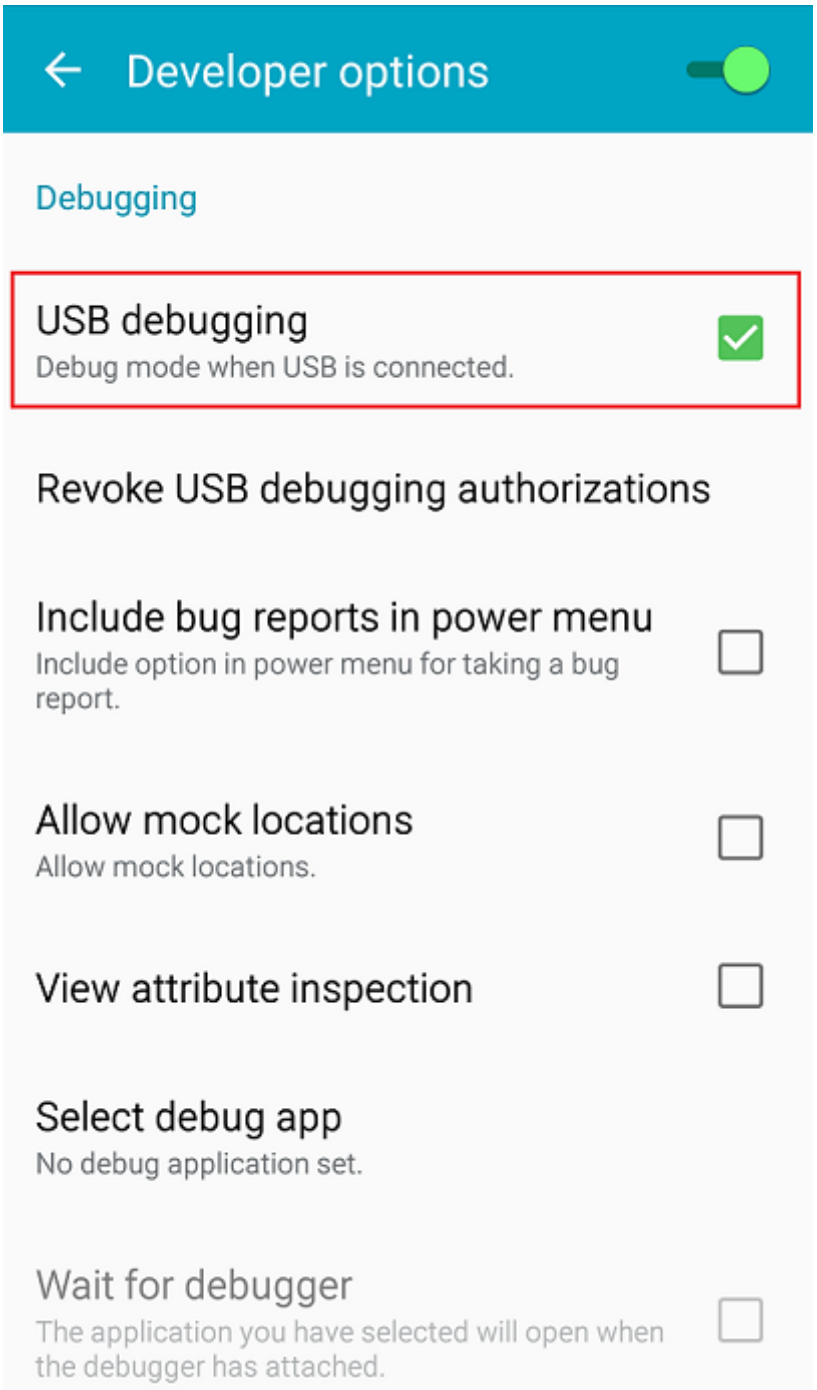
Note: Android Studio provides some ease of use benefits, but it is not fully tested for compatibility with Unity installs. If you encounter errors, Unity recommends using the command line method.

When installing the Android platform SDK and other tools, you can typically install the latest available version. There might be cases in which Google has released a new version of the SDK that causes errors when you build your Unity Project. In that case you must uninstall the SDK and install an earlier version.

Install the associated Platform and Build tools at the same time. If you are running on Windows, install the USB device drivers.

3. Enable USB debugging on your device

To enable USB debugging, you must enable Developer options on your device. To do this, find the build number in your device's **Settings** menu. The location of the build number varies between devices. The stock Android setting can be found by navigating to **Settings** > **About phone** > **Build number**. For specific information on your device and Android version, refer to your hardware manufacturer.



Build number as displayed in Android 5.0 (Lollipop) on a Samsung Galaxy Note 3

Note: On Android versions prior to 4.2 (Jelly Bean), the Developer options aren't hidden. Go to **Settings** > **Developer options**, then enable USB debugging.

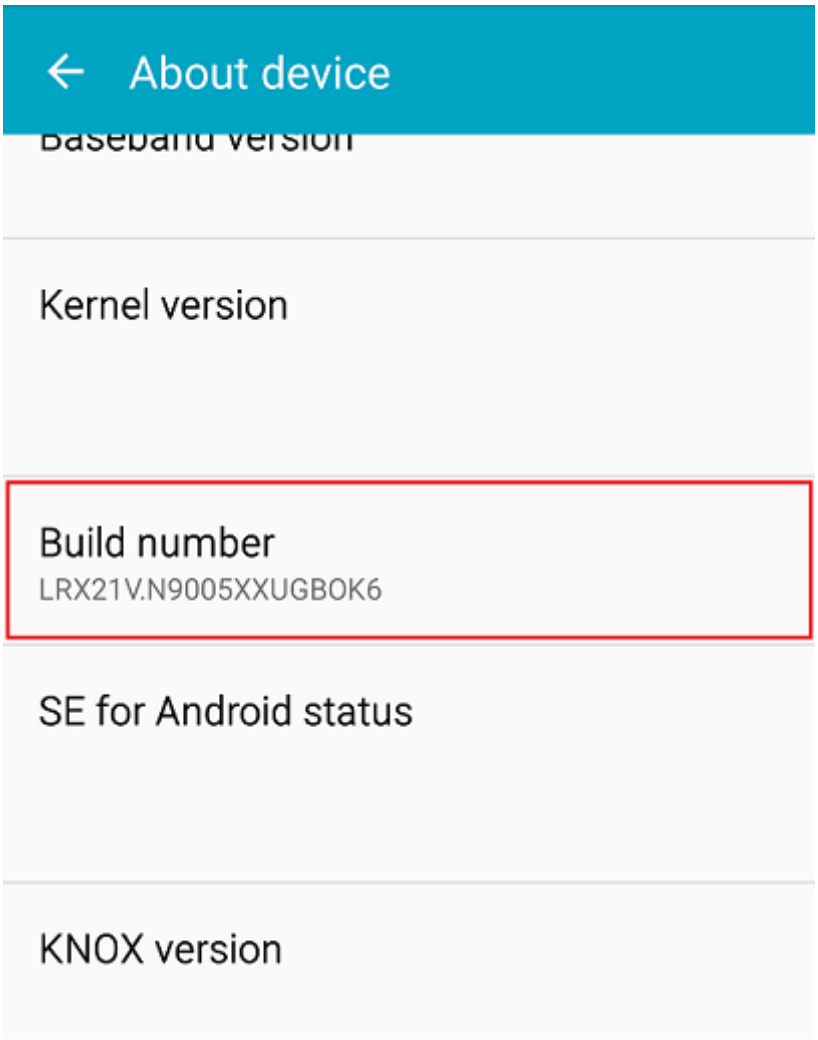
After you navigate to the build number using the instructions above, tap on the build number seven times. A pop-up notification saying "You are now X steps away from being a developer" appears, with "X" being a number that counts down with every additional tap. On the seventh tap, Developer options are unlocked.

Connect your device to your computer using a USB cable. If you are developing on a Windows computer, you might need to install the a device specific USB driver. See the manufacture web site for your device for additional information.

The setup process differs for Windows and macOS, and is explained in detail on the Android developer website. For more information on connecting your Android device to the SDK, refer to the Running Your App section of the

Android Developer documentation.

Go to **Settings** > **Developer options**, and check the USB debugging checkbox to enable debug mode when the device is connected to a computer via USB.



Developer options as displayed in Android 5.0 (Lollipop) - Samsung Galaxy Note 3

4. Configure the Android SDK path in Unity

The first time you create a Project for Android (or if Unity later fails to locate the SDK), Unity asks you to locate the folder in which you installed the Android SDK.

If you installed the SDK using the sdkmanager, you can find the folder in <android tools install location>\platforms\<android sdk folder>.

Example:

c:\<android tools install location>\platforms\android–27

If you installed the SDK when you installed Android Studio, you can find the location in the Android Studio SDK Manager. To open the SDK Manager from Android Studio, click **Tools > Android > SDK Manager** or click **SDK Manager** in the toolbar.

SDK Manager toolbar button

To change the location of the Android SDK, in the menu bar go to **Unity** > **Preferences** > **External Tools**.

5. Download and set up the Android NDK

If you are using the IL2CPP **scripting backend** for Android, you need the Android Native Development Kit (NDK). It contains the toolchains (such as compiler and linker) needed to build the necessary libraries, and finally produce the output package (APK). If you are not targeting the IL2CPP back end, you can skip this step.

Download Android NDK version r16b (64-bit) from the NDK Downloads web page. Extract the *android-ndk-r16b* folder to a directory on your computer and note the location.

The first time you build a Project for Android using IL2CPP, you are asked to locate the folder in which you installed the Android NDK. Select the root folder of your NDK installation. To change the location of the Android NDK, in the Unity Editor, navigate to the menu: **Unity** > **Preferences** to display the Unity Preferences dialog box. Here, click **External Tools**.

Android Remote (DEPRECATED)

Leave feedback

Android Remote is a Android application that makes your device act as a remote control for the project in Unity. This is useful for rapid development when you don't want to compile and deploy your project to device for each change.

DEPRECATED

Unity has replaced Android Remote with the new Unity Remote 5.

How to use Android remote

To use Android Remote, you should firstly make sure that you have the latest Android SDK installed (this is necessary to set up port-forwarding on the device). Then, connect the device to your computer with a USB cable and launch the Android Remote app. When you press Play in the Unity editor, the device will act as a remote control and will pass accelerometer and touch input events to the running game.

Troubleshooting Android development

Leave feedback

While developing for Android using Unity, you could encounter a number of issues. Problems are often related to plug-ins or incorrect **Project settings**. This section outlines the most common scenarios and relavant troubleshooting advice.

Your application crashes immediately after launch

Remove any native plug-ins you have.

Disable stripping.

Use `adb logcat` to get the crash report from your device. Consult the official Android Developer Logcat Command-Line Tool documentation for more information.

The game crashes after a couple of seconds when playing video

Ensure **Settings** > **Developer Options** > **Don't keep activities** isn't enabled on the device.

The video player is its own activity, and therefore regular game activity will be destroyed if the video player is activated.

No Android device found

If Unity cannot find an Android device connected to the system, check the following:

Make sure that your device is actually connected to your computer - check the USB cable and the sockets.

Make sure that your device has **USB Debugging** enabled in the Developer options. For more details, refer to the Android SDK/NDK Setup page.

Run the `adb devices` command from the `platform-tools` directory of your Android SDK installation and check the output.

If the output list is empty and you are using Windows, you may need to install the driver for **ADB** devices. For more details, refer to the Android SDK/NDK Setup documentation.

If the list contains entries with the **unauthorized** label, you may need to authorize your computer on your device and give it permission to debug it. Check the device's screen for the corresponding dialog.

If the list contains your device with the **device** label, build your Project in Unity again.

Failed to re-package resources

This error occurs when the Android **Asset** Packaging Tool (AAPT) fails. AAPT is used to build the intermediate **Asset packages** during Android build. This issue is most often caused by missing resources or duplicate resources in your Android plug-ins.

Check the console message for more details - it should contain the IDs of the resources that are missing or duplicates. Fix the error in your plug-ins by either adding the missing resources/settings or removing the duplicate plug-ins.

Unable to merge Android manifests

The most likely cause for this issue is that one of your plug-ins has a manifest that is incompatible with the main Unity manifest.

Check the console message for more details on which attributes are conflicting, and fix the manifests accordingly.

See the Android Manifest documentation for more details on Android manifests.

Unable to convert classes into DEX format

The most likely cause for this issue is that you have a Java plug-in added twice. This results in duplicate classes when Unity tries to build a DEX (Dalvik Executable Format) file from all the compiled Java plug-ins. Check the console output for the list of duplicate entries, and fix the plug-ins.

If your console messages says “Too many references”, it means that the number of fields and methods exceeded the DEX limit of 64k. This usually happens when the number of plug-ins or plug-in resources is too high. Due to the way the references are generated, the limit could be hit with just a couple of large plug-ins.

There are several ways to handle this issue. One of these is by stripping the plug-ins. However, the quickest way to fix it is to switch to the Gradle build system, or export the Project and build it in Android Studio.

Unable to install APK to device

This error can be caused by:

Installing to an incompatible device.

Installing to a device running a version of Android lower than the **Minimum API Level** in your **Player settings**.

Check the console for the actual error code and output.

2017–05–25 Page published with editorial review

Updated functionality in 5.5

Building apps for Android

Leave feedback

There are two locations to configure settings that affect how your app is built:

Player Settings - Allows you to configure runtime settings for the app. For more information on Player Settings, see Android Player Settings.
Build settings - Allows you to configure build system parameters and build the app.
The output package includes an **APK**, and an **APK** expansion file (OBB) if the **Split Application Binary** option is selected in the Unity Editor's Player Settings window. For more information on OBB files, see OBB Support.

To optimize for download and installation size, enable the **Split APKs by target architecture** option in **Player Settings**. The **Split APKs by target architecture** option produces one set of APKs and OBBs for each device architecture selected in the **Target Architecture** list in the **Player Settings**. You can upload this set of APKs (and OBBs, if enabled) to the Google Play, or other, store instead of a FAT APK in which all of the selected architectures are included into a single APK. For more information on this feature, see Multiple APK support on the Android Developer website.

Configuring Build Settings

To configure and build apps for Android, access the Build Settings window, select **File** > **Build Setting**. In **Platforms**, select **Android**.

To set Android as your default build platform, click the **Switch Platform** button.

When you have specified your build settings, click the **Build** button to create your build. To build the app, click **Build And Run** to create and run your build on the platform you have specified. In Platform, select **Android**.

Option	Purpose
Texture **Compression**	The Unity Android build system supports the following texture **compression** format options: Don't override, DXT (Tegra), **PVRTC** (PowerVR), **ETC** (default), ETC2 (GLES 3.0), and **ASTC**. For advice on using these formats, see the Texture Compression section.
ETC2 fallback	32-bit, 16-bit, and 32-bit, half resolution.
Build System	
	Internal - Generate the output package (APK) using the internal Unity build process, based on Android SDK utilities. Selecting Internal hides the Export Project checkbox.
	Gradle - Generate the output package (APK) using the Gradle build system. Supports direct Build and Run and exporting the Project to a directory. This is the default Build System for Unity.
Export Project	Export the Project as a Gradle project that you can import into Android Studio.
Development Build	A development build includes debug symbols and enables the **Profiler**. Selecting Development Build allows you to select the Autoconnect **Profiler**, Script Debugging, and **Scripts** Only Build options.
Autoconnect Profiler	Allows the Profiler to automatically connect to the build. Selectable when the Development Build option is selected. For more information on the Profiler, see Profiler Overview.
Script Debugging	Allow script debuggers to attach to the Player remotely. Enabled when the Development Build option is selected.
Scripts Only Build	Check this option to build just the scripts in the current Project. Enabled when the Development Build option is selected.
Compression Method	Compress the data in your Project at build time. Choose between the following methods:
	Default - The default compression is ZIP, which gives slightly better compression results than LZ4 and LZ4HC, but data is slower to decompress.
	LZ4 - A fast compression format that is useful for development builds. Using LZ4 compression can significantly improve loading time of games/apps built with Unity. For more information, see BuildOptions.CompressWithLz4.

Option	Purpose
	LZ4HC - A high compression variant of LZ4 that is slower to build but produces better results for release builds. Using LZ4HC compression can significantly improve loading time of games/apps built with Unity. For more information, see BuildOptions.CompressWithLz4HC.
SDKs for App Stores	Select which third party app stores to integrate with. To include an integration, click Add next to an App Store name. The Unity Package Manager automatically downloads and includes the relevant integration package.

Texture compression

Unity uses the Ericsson Texture Compression (ETC) format for textures that don't have individual **texture format** overrides. When building an APK to target specific hardware, use the **Texture Compression** option to override this default behavior. **Texture Compression** is a global setting for the Project. If a texture has a specific override on it, that texture is not affected by the **Texture Compression** setting. For additional information, see Textures.

For additional information on textures and texture compression, see Android 2D Textures Overrides.

For additional information on the texture compression formats, see Texture compression formats for platform-specific overrides. In particular, see the **Notes on Android** at the end of the topic.

Note: **Texture Compression** is a global setting. Individual textures override the global setting.

ETC2 fallback

For Android devices that don't support ETC2 (which don't support GL ES3), you can override the default ETC2 texture decompression by choosing from 32-bit, 16-bit, or 32-bit with half the resolution formats.

This option allows you to choose between the uncompressed image quality and the amount of memory the uncompressed texture occupies. 32-bit RGBA texture is the highest quality format, and takes twice the required disk space as the 16-bit format, but a texture in 16-bit might lose some valuable color information. 32-bit half-resolution reduces the memory requirement further, but the texture is likely to become blurry.

Build system

Unity supports two Android build systems: **Gradle** and **Internal**.

The steps involved with building for Android are:

Preparing and building the Unity **Assets**.

Compiling scripts.

Processing the plug-ins.

Splitting the resources into the parts that go to the APK and the OBB, if **Split Application Binary** is selected.

Building the Android resources using the AAPT utility (internal build only.)

Generating the Android manifest.

Merging the library manifests into the Android manifest (internal build only.)

Compiling the Java code into the Dalvik Executable format (DEX) (internal build only.)

Building the IL2CPP library, if **IL2CPP Scripting Backend** is selected.

Building and optimizing the APK and OBB packages.

Gradle build system

The Gradle build system uses Gradle to build an APK or export a Project in Gradle format, which can then be imported to Android Studio. When you select this build system, Unity goes through the same steps as the **Internal** build system excluding resource compilation with AAPT, merging manifests, and running DEX. Unity then generates the *build.gradle* file (along with the other required configuration files) and invokes the Gradle executable, passing it the task name and the working directory. Finally, the APK is built by Gradle.

For more details, see Gradle for Android.

Internal build system

The **Internal** build system creates an APK using the Android SDK utilities to build and optimize the APK and OBB packages. For more information about OBB files, see OBB Support.

Exporting the Project

If you need more control over the build pipeline, or to make changes that Unity does not normally allow (for example, fine tuning the manifest files that are automatically generated by Unity), you can export your Project and import it into Android Studio. Exporting a Project is only available when you have selected **Gradle** as your **Build System**.

To export the Project:

From the **Build System** drop-down menu, select **Gradle**.
Check the **Export Project** checkbox. When **Export Project** is checked, the **Build** button is relabeled **Export** and the **Build And Run** button is disabled.
Click the **Export** button and select a destination folder for the roject.
When the export finishes, open Android Studio and import your project. For more information on importing projects to Android Studio, see the Migrate to Android Studio section of the Android Developer documentation.

Build or Build and Run

The **Build Settings** window offers two options: **Build** and **Build and Run**. Using either option saves the output packages (APK and OBB, if enabled) to the path that you select. You can publish these packages to the Google Play Store, or install them on your device manually with the help of Android Debug Bridge (ADB). For further information about installing apps manually, see the Run your app section of the Android Developer documentation. For information on **ADB** commands, see the Android Debug Bridge section of the Android Developer documentation.

Selecting **Build and Run** saves the output packages to the file path you specify, while also installing your app on the Android device connected to your computer.

If the **Split Application Binary** option is enabled, the OBB file is pushed to the correct location on your device. If **Development Build** is checked, Unity also sets up a Profiler tunnel and enables **CheckJNI**. After that, the app is launched. The **Split Application Binary** setting is located in the **Publishing Settings** section of the **Player Settings** window.

Tip: Specify the output path for the packages and then use the **Ctrl+B** (Windows) or **Cmd+B** (macOS) keyboard shortcut to **Build and Run** using the saved output path.

2018–09–03 Page amended with editorial review

Updated functionality in 5.5

Reporting crash bugs under Android

Leave feedback

Before submitting a bug report, refer to the Troubleshooting Android development page of the Unity Manual and the Unity Android forum for solutions to common crashes and problems.

Investigating crashes on Android can be challenging, especially if you cannot reproduce the crash on your own device. use the Android Debug Bridge (ADB) to create a log from your device by running `adb logcat`. Information about what the **ADB** logcat file contains can be found on the logcat command line tool page of the official Android Studio documentation.

If you are unable to find a solution to your problem through the above methods, submit a bug report and attach the ADB logcat file and the full Unity Project to your bug report before submitting, following the Unity bug reporting guidelines.

2017–05–25 Page published with editorial review

Updated functionality in 5.5

Support for APK expansion files (OBB)

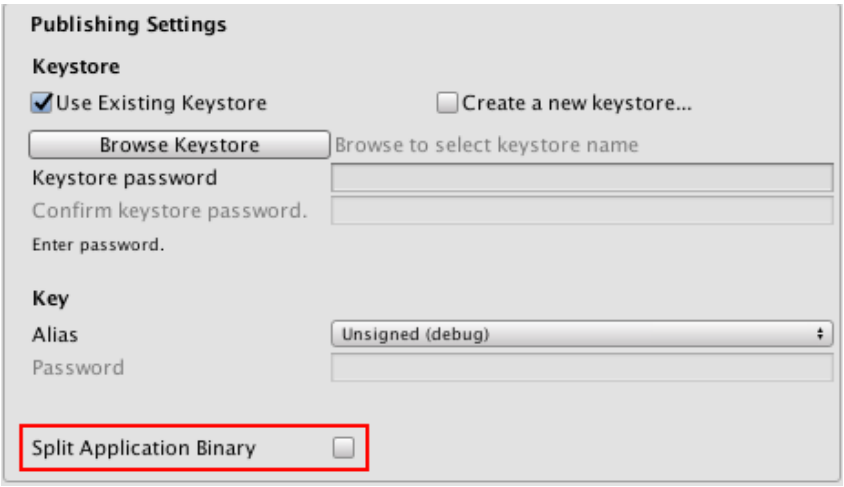
Leave feedback

APK expansion files are used as a solution for the 100MB app size limit in the Google Play Store. If your app is larger than 100MB (which is quite likely for a big game), you have to split your output package into the main part (APK) and the expansion file (OBB). Refer to the Android Developer documentation on expansion files for more information.

Unity automatically splits the output package into APK and OBB. This is not the only way to split the app package (other options include third-party plug-ins and AssetBundles), but it is the only automatic splitting mechanism officially supported by Unity.

Building the app with expansion files

If you want Unity to split the app output package into APK and OBB for you, open the **Player Settings** window (menu: **Edit** > **Project Settings** > **Player**), and in the **Publishing Settings** section, tick the **Split Application Binary** checkbox.



The **Publishing Settings** section of the **Player Settings** window, with the **Split Application Binary** checkbox highlighted

Both parts of the output package (APK and OBB) are copied to the output directory you specify when building the app. For example, if the APK has the name *mygame.apk*, the OBB is in the same directory under the name *mygame.main.obb*.

If you select **Build and Run**, the APK and OBB files are installed on your device by Unity. If you select **Build** and want to install the app manually using the **ADB** utility, you must first install the APK and then copy the OBB into the correct location on your device. The OBB file name must correspond the format required by Google. Refer to the expansion files section of the Android Developer documentation for more information.

If the app starts and can't find and load the OBB, only the first **Scene** is available (see documentation on how data is split between the APK and OBB below for more information). Do not use the contents of the OBB separately - always treat the APK and OBB as a unique bundle, the same way as you would treat a single APK.

How data is split between the APK and OBB

When the **Split Application Binary** option is enabled, the app is split the following way:

APK - Consists of the executables (Java and native), plug-ins, **scripts**, and the data for the first Scene (with the index 0).

OBB - Contains everything else, including all of the remaining Scenes, resources, and streaming **Assets**.

If your APK is still too large for publishing in the Google Play Store (more than 100MB), try reducing the size of your first Scene, making it as small as possible.

Downloading the OBB expansion file

The Unity Asset Store offers a plug-in that allows you to access an adapted version of the Google Play `market_downloader` library for Unity, which you can use to download the OBB from Google Play Store, or an external source, and move it into the correct directory.

Hosting OBB files on the Google Play Store

OBB expansion files should be published to the Google Play Store along with your APK. Any OBB files published with your APK will be automatically downloaded when a user installs your app from the Google Play Store.

You should include code in your app that downloads missing OBB files in the case of a Google Play Store error, or if a user removes the OBB files from their device. For more information about downloading OBB files, refer to the APK Expansion file section of the Android Developer documentation.

Hosting OBB files without using the Google Play Store

You can also host OBB files yourself if you do not want to use the Google Play Store. However, hosting OBB files without using the Google Play Store is only recommended for advanced users.

2017–05–25 Page published with editorial review

Updated functionality in 5.5

Android Scripting

Leave feedback

Most features of the Android devices are exposed through the Input and Handheld classes. For cross-platform projects, UNITY_ANDROID is defined for conditionally compiling Android-specific C# code.

For further information, see the Input, Mobile Keyboard and Device Property pages.

Advanced Unity mobile scripting

Leave feedback

Device properties

There are a number of device-specific properties that you can access when developing for mobile. See the Unity Scripting API pages for SystemInfo.deviceUniqueIdentifier, SystemInfo.deviceName,SystemInfo.deviceModel, and SystemInfo.operatingSystem for more information.

Anti-piracy check

Pirates often hack an app by removing DRM protection and then redistributing it for free. Unity comes with an anti-piracy check, which is used to determine if your app was altered after it was submitted to the Google Play Store.

To check if your app is genuine (meaning it is not hacked), use the Unity Scripting API Application.genuine property. If this property returns `false`, you can inform the app user that they are using a hacked app or you can disable access to some functions of your app.

Note: Use Application.genuineCheckAvailable along with Application.genuine to verify app integrity. Accessing the Application.genuine property is a resource-intensive operation, so do not call it during frame updates or other time-critical code.

Vibration support

To trigger a vibration, call Handheld.Vibrate in your code. Devices lacking vibration hardware ignore this call.

Activity indicator

Mobile operating systems have built-in activity indicators that you can activate during slow operations.

Refer to the Activity Indicator Unity Scripting API documentation for examples.

Screen orientation

When creating Projects for **iOS** and Android, you can control the screen orientation of a user's device. Detecting a change in orientation or forcing a specific orientation is useful for altering game behavior depending on how the user is holding their device.

To retrieve device orientation, use the Screen.orientation property. Orientation can be one of the following:

Orientation	Behavior
Portrait	The device is in portrait mode, with the device held upright and the home button at the bottom.
PortraitUpsideDown	The device is in portrait mode but upside-down, with the device held upright and the home button at the top.
LandscapeLeft	The device is in landscape mode, with the device held upright and the home button on the right side.
LandscapeRight	The device is in landscape mode, with the device held upright and the home button on the left side.

Control screen orientation by setting Screen.orientation to one of the above options, or to ScreenOrientation.AutoRotation.

When using auto-rotation, you can disable some orientations on a case-by-case basis. See the Scripting API pages for Screen.autorotateToPortrait, Screen.autorotateToPortraitUpsideDown, Screen.autorotateToLandscapeLeft, and Screen.autorotateToLandscapeRight for more information.

2017–05–25 Page published with editorial review

Updated functionality in 5.5

Building and using plug-ins for Android

Leave feedback

You can use plug-ins developed for Android to call Java and C++ code created outside of Unity from your C# **scripts**, allowing you to access features like OS calls and third-party code libraries that would otherwise not be available to Unity. For more information about using plug-ins with Unity, see documentation on plug-ins.

The following pages describe the process of creating your own plug-ins for use in Android projects. The information on these pages assumes the reader knows how to create native plug-ins for use in Unity. For more information about native plug-ins and their uses, see documentation on native plug-ins

Unity supports several types of Android plug-ins:

AAR plug-ins and Android Libraries

JAR plug-ins

Extending the UnityPlayerActivity Code

Native (C++) plug-ins

2017–05–18 Page published with editorial review

Updated features in 5.5

AAR plug-ins and Android Libraries

Leave feedback

AAR plug-ins

Android Archive (AAR) plug-ins are bundles that include compiled Java and native (C/C++) code, resources, and an Android Manifest. The .aar file itself is a zip archive which contains all of the **Assets**. For more details, see Android Developer documentation on creating an Android Library.

To add an AAR plug-in to your project, copy the .aar file into any of your project folders, then select it in Unity to open the Import Settings in the **Inspector** window. Tick the **Android** checkbox to mark this .aar file as compatible with Unity:

ARR plug-in import settings as displayed in the Inspector window

AAR is the recommended plug-in format for Unity Android applications.

Android Library Projects

Android Library projects are similar to AAR plug-ins: they contain native and Java code, resources, and an Android Manifest. However, an Android Library is not a single archive file, but a directory with a special structure which contains all of the Assets. For more details, see Android Developer documentation on creating an Android Library.

Import pre-compiled Android library projects into the Assets/**Plugins**/Android folder. Pre-compiled means that all .java files must have been compiled into .jar files and placed in either the bin/ or the libs/ folder of the Android Studio project before being imported into Unity. From these folders, AndroidManifest.xml gets automatically merged with the main manifest file when the project is built.

Unity treats any subfolder of Assets/Plugins/Android as a potential Android Library, and disables Asset importing from within these subfolders. The subfolder is recognized as an Android Library if it contains the AndroidManifest.xml file, and the project.properties file contains the string `android.library=true`.

See Android Developer documentation on the Library module for more details.

Providing additional Android Assets and resources

If you need to add Assets to your Unity application that should be copied unchanged into the output package, import them into the Assets/Plugins/Android/assets directory. They appear in the assets/ directory of your **APK**, and are accessed by using the getAssets() Android API from your Java code.

2017–05–18 Page published with editorial review

Updated features in 5.5

JAR plug-ins

Leave feedback

JAR(Java Archive) plug-ins are primarily used to enable interaction with the Android OS or to call methods written in Java from within your C# **scripts**.

They can only contain Java code (for example, they can't contain Android resources), which makes their use very limited. To add a JAR plug-in to your project, copy the .jar file into any of your project folders, then select it in Unity to open the Import Settings in the **Inspector** window. Tick the Android checkbox to mark this .jar file as compatible with Android:

JAR plug-in import settings as displayed in the Inspector window

Using Java plug-ins

Unity uses the Java Native Interface (JNI) both when calling code from Java and when interacting with Java or the Java VM(Virtual Machine) from native code or C# scripts.

Java source files as plug-ins

When using the Gradle build system, you can avoid creating JAR files. To do this:

Drop the .java file into your Unity project as a plug-in.
In the Plugin Inspector, mark the plug-in for the Android platform.
When building for Android, make sure **Gradle** is set as your **Build System** in the Build Settings.
Unity copies the java file to the Gradle project and builds with it.

Using your Java plug-in from native (C/C++) code

Note: This information in this section requires advanced knowledge of the Android Java Native Interface (JNI).

To access your Java code from C or C++ plug-ins, you need access to the Java VM. Add the following method to your C/C++ code to access the Java VM.

```
jint JNI_OnLoad(JavaVM* vm, void* reserved) {
  JNIEnv* jni_env = 0;
  vm->AttachCurrentThread(&jni_env, 0);
  return JNI_VERSION_1_6;
}
```

It is beyond the scope of this document to explain JNI completely, but this method usually involves finding the class definition, resolving the constructor (<init>) method and creating a new object instance, as shown in this example:

```
jobject createJavaObject(JNIEnv* jni_env) {
  jclass cls_JavaClass = jni_env->FindClass("com/your/java/Class");         // f
  jmethodID mid_JavaClass = jni_env->GetMethodID (cls_JavaClass, "<init>", "()V"
  jobject obj_JavaClass = jni_env->NewObject(cls_JavaClass, mid_JavaClass);
  return jni_env->NewGlobalRef(obj_JavaClass);                      // return ob
}
```

For more information about JNI, see Android Developer documentation on JNI

Using your Java plug-in from C# scripts with helper classes

Note: This information in this section requires advanced knowledge of the Android Java Native Interface (JNI).

The `AndroidJNIHelper` and `AndroidJNI` Unity API classes are used as a wrapper around a “raw” JNI interface.

The AndroidJavaObject and AndroidJavaClass Unity API classes automate a lot of tasks when using JNI calls, and they also use caching to make calls to Java faster. The combination of `AndroidJavaObject` and `AndroidJavaClass` is built on top of `AndroidJNI` and `AndroidJNIHelper`, but also has some additional functionality. These classes also have static methods that are used to access static members of Java classes.

There are three ways to make Java JNI calls from your C# scripts:

raw JNI through the `AndroidJNI` methods ;
`AndroidJNIHelper` class together with `AndroidJNI`;
`AndroidJavaObject` and `AndroidJavaClass` classes as the most convenient high-level APIs.
UnityEngine.AndroidJNI is a wrapper for the JNI calls available in C (as described above). All methods in this class are static and have a 1:1 mapping to the Java Native Interface.

UnityEngine.AndroidJNIHelper provides helper functionality used by the next level, which is exposed as public methods and might be useful in special cases.

Instances of UnityEngine.AndroidJavaObject and UnityEngine.AndroidJavaClass have a one-to-one mapping to an instance of java.lang.Object and java.lang.Class (or their subclasses) on the Java side, respectively. They essentially provide 3 types of interaction with the Java side:

Call a method

Get the value of a field

Set the value of a field

The call is separated into two categories: A call to a 'void' method, and call to a method with non-void return type. A generic type is used to represent the return type of those methods which return a non-void type. The Get and Set always take a generic type representing the field type.

Examples

Example 1

```
AndroidJavaObject jo = new AndroidJavaObject("java.lang.String", "some_string")
// jni.FindClass("java.lang.String");
// jni.GetMethodID(classID, "<init>", "(Ljava/lang/String;)V");
// jni.NewStringUTF("some_string");
// jni.NewObject(classID, methodID, javaString);
int hash = jo.Call<int>("hashCode");
// jni.GetMethodID(classID, "hashCode", "()I");
// jni.CallIntMethod(objectID, methodID);
```

This example creates an instance of java.lang.String initialized with a string, and retrieves the hash value for that string.

The `AndroidJavaObject` constructor takes at least one parameter - the name of the class of to construct an instance of. Any parameters after the class name are for the constructor call on the object, in this case the string "some_string". The subsequent call to hashCode() returns an 'int' which is is used as the generic type parameter to the call method in this example.

Note: A nested Java class cannot be instantiated using dotted notation. Inner classes must use the $ separator. Use `android.view.ViewGroup$LayoutParams` or `android/view/ViewGroup$LayoutParams`, where LayoutParams class is nested in ViewGroup class.

Example 2

This example shows how to get the cache directory for the current application in C# without using plug-ins:

```
AndroidJavaClass jc = new AndroidJavaClass("com.unity3d.player.UnityPlayer");
 // jni.FindClass("com.unity3d.player.UnityPlayer");
 AndroidJavaObject jo = jc.GetStatic AndroidJavaObject>("currentActivity");
 // jni.GetStaticFieldID(classID, "Ljava/lang/Object;");
 // jni.GetStaticObjectField(classID, fieldID);
 // jni.FindClass("java.lang.Object");

 Debug.Log(jo.Call AndroidJavaObject>("getCacheDir").Call<string>("getCanonicalP
 // jni.GetMethodID(classID, "getCacheDir", "()Ljava/io/File;"); // or any basec
 // jni.CallObjectMethod(objectID, methodID);
 // jni.FindClass("java.io.File");
 // jni.GetMethodID(classID, "getCanonicalPath", "()Ljava/lang/String;");
 // jni.CallObjectMethod(objectID, methodID);
 // jni.GetStringUTFChars(javaString);
```

This example starts with `AndroidJavaClass` instead of `AndroidJavaObject` in order to access a static member of `com.unity3d.player.UnityPlayer` rather than create a new object. Then the static field "currentActivity" is accessed but this time `AndroidJavaObject` is used as the generic parameter. This is because the actual field type android.app.Activity is a subclass of `java.lang.Object`, and any non-primitive type must be accessed as `AndroidJavaObject`. The exceptions to this rule are strings, which are accessed directly even though they don't represent a primitive type in Java.

`getCacheDir()` can then be called on the Activity object to get the File object representing the cache directory, `getCanonicalPath()` can then be called to get a string representation.

Unity provides access to the application's cache and file directory with `Application.temporaryCachePath` and `Application.persistentDataPath` APIs.

Example 3

This example shows how to pass data from Java to Unity using `UnitySendMessage`.

```
using UnityEngine;

public class NewBehaviourScript : MonoBehaviour {
```

```
    void Start () {
        AndroidJNIHelper.debug = true;
        using (AndroidJavaClass jc = new AndroidJavaClass("com.unity3d.player.Un
        jc.CallStatic("UnitySendMessage", "Main Camera", "JavaMessage", "NewMess
        }
    }

    void JavaMessage(string message) {
        Debug.Log("message from java: " + message);
    }
}
```

The Java class `com.unity3d.player.UnityPlayer` has a static method `UnitySendMessage`, equivalent to the **iOS** method: `UnitySendMessage`. It is used in Java to pass data to Unity.

Although `UnitySendMessage` is called from within Unity, it relays the message using Java, then Java calls back to the native/Unity code to deliver the message to the object named "Main **Camera**". This object has a script attached which contains a method called `JavaMessage`.

Best practice when using Java plug-ins with Unity

`AndroidJavaObject` and `AndroidJavaClass` are computationally expensive methods (as are any methods that use raw JNI). Keep the number of transitions between managed and native/Java code to a minimum, for better performance and also code clarity.

```
//The first time you call a Java method like
AndroidJavaObject jo = new AndroidJavaObject("java.lang.String", "some_string");
int hash = jo.Call<int>("hashCode"); // first time - expensive
int hash = jo.Call<int>("hashCode"); // second time - not as expensive as we alr
```

The Mono garbage collector should release all created instances of `AndroidJavaObject` and `AndroidJavaClass` after use, but it is advisable to keep them in a `using(){}` statement to ensure they are deleted as soon as possible. Without this, you cannot be sure when they are be destroyed. If you set `AndroidJNIHelper.debug` to true, you will see a record of the garbage collector's activity in the debug output.

```
//Getting the system language safely
void Start () {
    using (AndroidJavaClass cls = new AndroidJavaClass("java.util.Locale")) {
        using(AndroidJavaObject locale = cls.CallStatic<AndroidJavaObject>("getD
```

```
            Debug.Log("current lang = " + locale.Call<string>("getDisplayLanguag

        }
    }
}
```

2018–03–20 Page published with no editorial review

Updated features in 5.5

Support for Java source file plug-ins added for Android player in 2018.2

Extending the UnityPlayerActivity Java Code

Leave feedback

When developing a Unity Android application, it is possible to extend the standard UnityPlayerActivity class (the primary Java class for the Unity Player on Android, similar to AppController.mm on Unity iOS) by using plug-ins. An application can override any and all of the basic interaction between the Android OS and the Unity Android application.

Two steps are required to override the default activity:

Create the new Activity which derives from UnityPlayerActivity;

Modify the Android Manifest to have the new activity as the application's entry point.

The easiest way to achieve this is to export your project, and make the necessary modifications to the UnityPlayerActivity class in Android Studio.

To make a plug-in with your new activity code and add it to your Unity project you must perform the following steps:

Extend the UnityPlayerActivity. The UnityPlayerActivity.java file is found at */Applications/Unity/Unity.app/Contents/PlaybackEngines/AndroidPlayer/src/com/unity3d/player* on Mac and *C:\Program Files\Unity\Editor\Data\PlaybackEngines\AndroidPlayer\src\com\unity3d\player* on Windows. To extend the UnityPlayerActivity locate the classes.jar included with Unity. It is found in the installation folder (usually C:\Program Files\Unity\Editor\Data (on Windows) or /Applications/Unity (on Mac)) in a subfolder called PlaybackEngines/AndroidPlayer/Variations/mono or **il2cpp**/Development or Release/Classes/. Then add classes.jar to the classpath used to compile the new Activity. Compile your Activity source file and package it into a JAR or AAR package, and copy it to your project folder..

Create a new Android Manifest to set the new activity as the entry point of your application. Place the AndroidManifest.xml file in the **Assets**/**Plugins**/Android folder of your project.

The following is an example UnityPlayerActivity file:

```
OverrideExample.java:
package com.company.product;
import com.unity3d.player.UnityPlayerActivity;
import android.os.Bundle;
import android.util.Log;

public class OverrideExample extends UnityPlayerActivity {
  protected void onCreate(Bundle savedInstanceState) {
    // call UnityPlayerActivity.onCreate()
    super.onCreate(savedInstanceState);
    // print debug message to logcat
    Log.d("OverrideActivity", "onCreate called!");
  }
```

```
  public void onBackPressed()
  {
    // instead of calling UnityPlayerActivity.onBackPressed() we just ignore the
    // super.onBackPressed();
  }
}
```

```
And this is what the corresponding AndroidManifest.xml could look like:
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android" package="co
  <application android:icon="@drawable/app_icon" android:label="@string/app_name
    <activity android:name=".OverrideExample"
             android:label="@string/app_name"
             android:configChanges="fontScale|keyboard|keyboardHidden|locale|mnc
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
  </application>
</manifest>
```

2017–05–18 Page published with no editorial review

Updated features in 5.5

Native (C++) plug-ins for Android

Leave feedback

Unity supports native plug-ins for Android written in C/C++ and packaged in a shared library (.so) or a static library (.a). When using the **IL2CPP scripting backend**, you can use C/C++ source files as plug-ins and Unity compiles them along with **IL2CPP** generated files. This includes all C/C++ source files with extensions .c, .cc, .cpp and .h.

To build a C++ plug-in for Android, use the Android NDK and get yourself familiar with the steps required to build a shared library. The same applies to static libraries.

If you are using C++ to implement the plug-in, you must ensure the methods are declared with C linkage to avoid name mangling issues. By default, only the C source files that have a .c file extension in the plug-ins have C linkage (not C++).

```
extern "C" {
  float Foopluginmethod ();
}
```

After building the library, copy the output .so file(s) into the **Assets/**Plugins**/Android** directory in your Unity project. In the **Inspector**, mark your .so files as compatible with Android, and set the required CPU architecture in the dropdown box:

Native(C++) plug-in import settings as displayed in the Inspector window

To call the methods in your native plug-in from within your C# **scripts**, use the following code:

```
[DllImport ("pluginName")]
private static extern float Foopluginmethod();
```

Note that pluginName should not include the prefix ('lib') or the extension ('.so') of the filename. It is recommended to wrap all the native plug-in method calls with an additional C# code layer. This code checks Application.platform and calls native methods only when the app is running on the actual device; dummy values are returned from the C# code when running in the Editor. Use platform defines to control platform dependent code compilation.

When you use C/C++ source files as plug-ins, you call them from C# in the same way except that you use `__Internal` for plug-in name, for example:

```
[DllImport ("__Internal")]
private static extern float Foopluginmethod();
```

Native (C++) plug-in Sample

This zip archive contains a simple example of a native code plug-in. This sample demonstrates how C++ code is invoked from a Unity application. The package includes a **scene** which displays the sum of two values as calculated by the native plug-in. You will need the Android NDK to compile the plug-in.

2018–03–10 Page published with limited editorial review

Updated features in 5.5

Support for using C++ source files and static libraries as plug-ins on Android added in 2018.2

Using Java source files as plug-ins

Leave feedback

Unity supports plug-ins for Android written in Java. This functionality allows you to add .java files into your Unity Project and they are compiled when you build an Android player. Previously you had to precompile these files as an android library.

Gradle is the only build system that supports this feature so use **Gradle** as your build system.

To compile Java source files as a plug-in:

Open the Unity Editor.
Select the Project to add the plug-in to.
In the Projects window, select the **Assets** folder and create a folder for your plug-in.
Drag and drop your Java source files into the folder.
Select each of the files.
In the **inspector** window, under **Select platforms for plugin**, verify that only **Android** is selected.
Build your Project.

Note: You can place your Java source files in any folder in your Project except in special use locations such as subfolders under Assets/**Plugins**/Android. If you place files in these locations, the Unity Editor will not display the plug-in inspector. For additional information, see AAR plug-ins and Android Libraries.

Use the AndroidJavaObject class to call methods in your plug-in. For additional information see, the “Using your Java plug-in from C# **scripts** with helper classes” section in JAR plug-ins.

2018–09–07 Page published with editorial review

Added plug-in support in 2018.2

Customizing an Android Splash Screen

Leave feedback

If you use Unity Personal Edition to make your project, a default splash screen displays while your game loads. It is oriented according to the **Default Screen Orientation** option in the Player Settings. You cannot change this splash screen.

Unity Professional Edition users can change the splash screen. You can use any texture in the project as a splash screen. You can set the texture from the Splash Image section of the Android Player Settings. You should also select the **Splash scaling** method from the following options:-

Center (only scale down) draws your image at its natural size unless it is too large, in which case it is scaled down to fit.
Scale to fit (letter-boxed) draws your image so that the longer dimension fits the screen size exactly. Empty space around the sides in the shorter dimension is filled in black.
Scale to fill (cropped) scales your image so that the shorter dimension fits the screen size exactly. The image is cropped in the longer dimension.

Single-Pass Stereo Rendering for Android

Leave feedback

Single-pass stereo **rendering** is currently supported on Gear VR and Daydream devices. To enable this feature we use the GL_OVR_multiview2 and GL_OVR_multiview_multisampled_render_to_texture OpenGL|ES extensions*. These extensions require the use of a texture 2D array with 2 slices (one slice per eye). This differs from our PC/**PS4** implementation where we just use a double wide 2D texture. Therefore different **shader** changes are required.

Shader script requirements

You may need to include the additional code listed below to use Single-Pass Stereo Rendering with your custom shaders. This doesn't apply to **surface shaders** or fixed function pipeline shaders which Unity automatically converts, except **surface shaders** that have custom vertex processing.

Note that these shader script changes are compatible with Multi-Pass stereo rendering, so there will be no side effects in that mode.

First, if you'd like to use `unity_StereoEyeIndex` in shader stages after the **vertex shader** you must declare `UNITY_VERTEX_OUTPUT_STEREO` in any shader stage output structs that you have. Here's an example:

```
struct v2f {
    float2 uv : TEXCOOR0;
    float4 vertex : SV_POSITION;
    UNITY_VERTEX_OUTPUT_STEREO
};
```

Next, you must use `UNITY_INITIALIZE_VERTEX_OUTPUT_STEREO()` in the vertex shader function to initialize the output data:

```
v2f vert (appdata v)
{
    v2f o;
    UNITY_INITIALIZE_VERTEX_OUTPUT_STEREO(o);
    o.vertex = UnityObjectToClipPos(v.vertex);
    o.uv = TRANSFORM_TEX(v.uv, _MainTex);
    return o;
}
```

To initialize `unity_StereoEyeIndex` in subsequent stages you need to add `UNITY_SETUP_STEREO_EYE_INDEX_POST_VERTEX()` at the beginning like so:

```
fixed4 frag (v2f i) : SV_Target
{
    UNITY_SETUP_STEREO_EYE_INDEX_POST_VERTEX(i);
    // sample the texture
    fixed4 col = tex2D(_MainTex, i.uv);
    // apply fog
    UNITY_APPLY_FOG(i.fogCoord, col);
    return col;
}
```

If you are using other shader stages you should also use the `UNITY_TRANSFER_VERTEX_OUTPUT_STEREO()` macro to transfer the eye index to the subsequent stages.

Finally, we recommend that you use `UnityObjectToClipPos(IN.vertex)` instead of mul `(UNITY_MATRIX_MVP, IN.vertex)` to calculate the final position of the object.

Post-Processing Shaders

Post-Processing shaders will need to be updated to deal with the eye textures being a texture 2D array. To help out with this we've created the following macro: `UNITY_DECLARE_SCREENSPACE_TEXTURE()`. This macro should be used to wrap all textures, so that the texture will work in both multi-pass and single-pass modes.

When sampling the texture the following macro should be used:

`UNITY_SAMPLE_SCREENSPACE_TEXTURE()`

This macro requires that `UNITY_SETUP_STEREO_EYE_INDEX_POST_VERTEX()` has already been called beforehand when in single-pass mode. We've also created similar macros for depth textures and screen space shadow maps. You can see the full list at the bottom of HLSLSupport.cginc.

See the Vertex and fragment shader examples page for more information on shader code.

*** These OpenGL|ES extensions are relatively new, so unfortunately there may be graphics related issues due to the drivers on these phones.**

New feature in 5.5

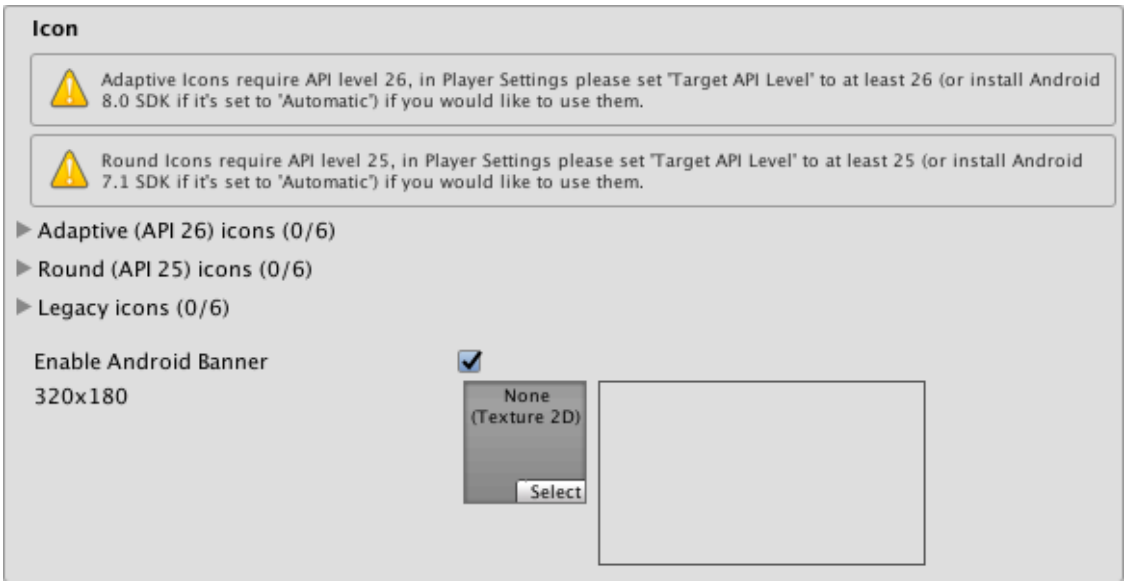
2017–06–20 Page published with limited editorial review

Android Player Settings

Leave feedback

This page details the **Player Settings** specific to Android. For a description of the general Player Settings, see documentation on PlayerSettings.

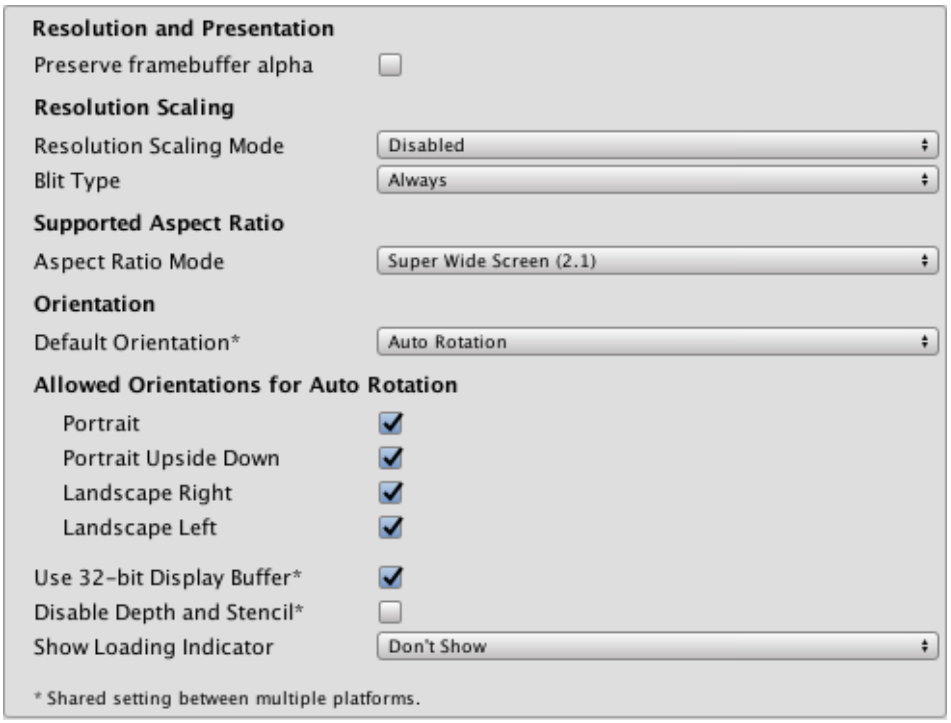
Icon



The **Icon** section of the Android PlayerSettings window

Property:	*Function:*
Adaptive	Set up textures for the Android Adaptive icons in your app.
Round	Set up textures for the Android Round icons in your app.
Legacy	Set up textures for the Android Legacy icons in your app.
Enable Android Banner	Enables a custom banner for Android TV builds.

Resolution And Presentation

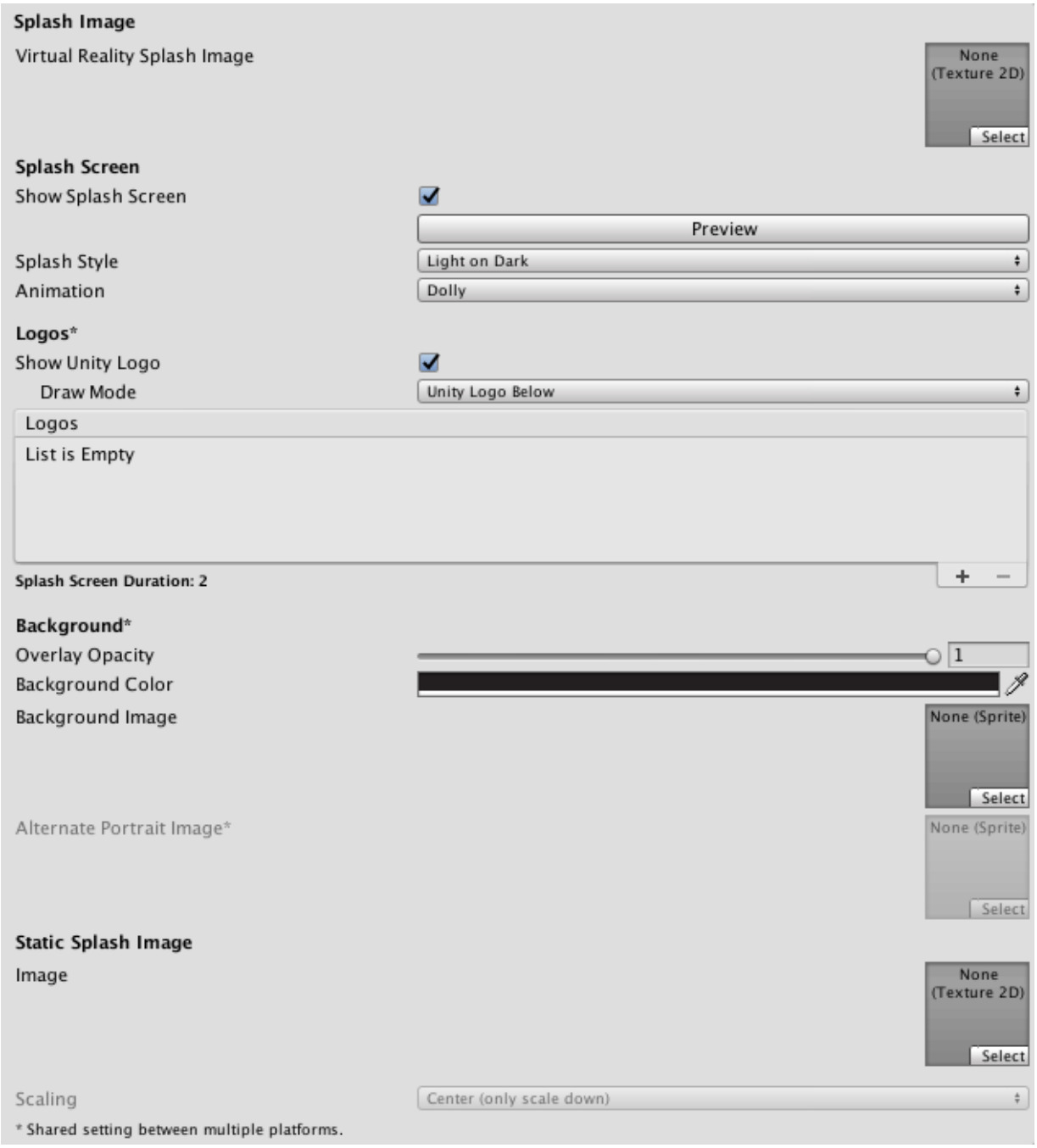


The **Resolution and Presentation** section of the Android PlayerSettings window

Property:	*Function:*
Resolution and Presentation	
Preserve framebuffer alpha	Enable Unity to render on top of native Android **UI**. The **camera**'s Clear Flags have to be set to Solid color with an alpha less than 1 for this to have any effect. (OpenGL ES only).
Resolution Scaling	
Resolution Scaling Mode	This option can be used to render the **scene** on the target device at a resolution lower than screen resolution.
Disabled	Render at full screen resolution.
Fixed DPI	Render at resolution potentially lower than screen resolution.
Target DPI	When Resolution Scaling Mode is set to Fixed DPI, this setting determines how much lower the target rendering resolution is going to be. The scale is calculated as min(Target DPI * Factor / Screen DPI, 1). Factor is controlled by Quality Settings.
Blit Type	Controls whether a blit is used to present the final image to the screen.
Always	(Always blit) Make Unity render to an offscreen buffer and then copy to the system framebuffer. This is compatible with most devices, but is usually slower than **Never** blitting.
Never	(Never blit) Make Unity render to the framebuffer provided by the OS. If a condition arises during application run time that causes this to fail, the application will print a one-time warning to the device log. **Never** blitting is usually faster than **Always** blitting, but it's not compatible with all devices.
Auto	Make Unity render to the framebuffer provided by the OS, if possible. If a condition is met that prevents your application from rendering to the system framebuffer, the application switches to offscreen rendering and issues a warning to the device console.
Support Aspect Ratio	
Aspect Ratio Mode	The aspect ratio for the device. Choose from: **Legacy Wide Screen (1.86)**, **Super Wide Screen (2.1)** and **Custom**. When you select **Custom**, an **Up To** field appears. Use this to set a maximum screen width.
Orientation	
Default Orientation	Set the application's screen orientation.
Portrait	Orient the screen so that the device's Home button is at the bottom.
Portrait Upside Down	Orient the screen so that the device's Home button is at the top.
Landscape Left	Orient the screen so that the device's Home button is on the right-hand side.
Landscape right	Orient the screen so that the device's Home button is on the left-hand side.
Allowed Orientations for Auto Rotation	
(Only visible when **Default Orientation** is set to **Auto Rotation**)	
Portrait	Allow portrait orientation.
Portrait Upside Down	Allow portrait upside-down orientation.

Property:	*Function:*
Landscape Right	Allow landscape right orientation (home button is on the left side).
Landscape Left	Allow landscape left orientation (home button is on the right side).
Other	
Use 32-bit Display Buffer	Specifies if the Display Buffer should be created to hold 32-bit color values (16-bit by default). Use it if you see banding, or need alpha in your post-processed effects, because they create Render Textures in the same format as the Display Buffer.
Disable Depth and Stencil	Tick this checkbox to disable the depth and stencil buffers.
Show Loading Indicator	The type of loading progress indicator that should be shown. Options are **Don't Show**, **Large**, **Inversed Large**, **Small** and **Inversed Small**.

Splash Image

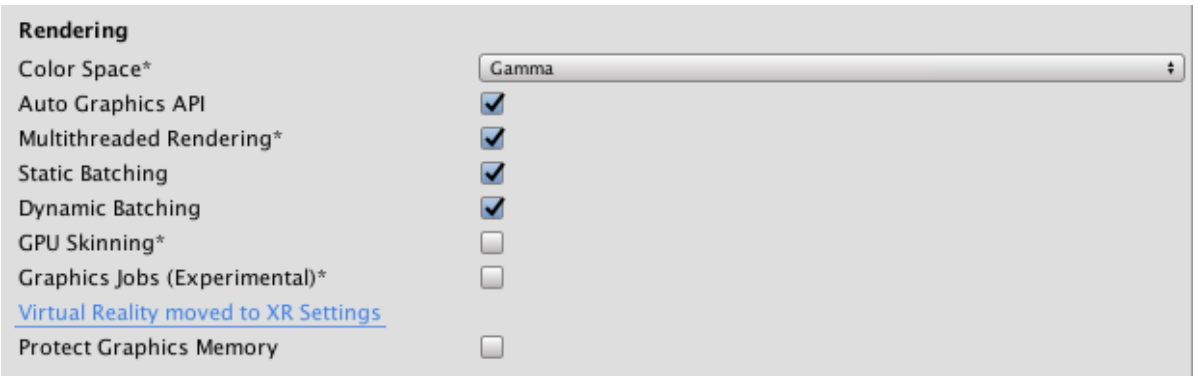


Property:	*Function:*
Virtual Reality Splash Screen	Specify the texture that should be used for the Android splash screen on a **Virtual Reality** application.
Android Splash Screen	Specify the texture that should be used for the Android splash screen. The standard size for the splash screen image is 320x480.
Splash Scaling	Specify how the splash image is scaled to fit the device's screen. The options are **Center (only scale down)**, **Scale to Fit (letter-boxed)** and **Scale to Fill (cropped)**.

See also Unity Splash Screen settings.

Other Settings

Rendering



The **Rendering** section of the Android PlayerSettings window

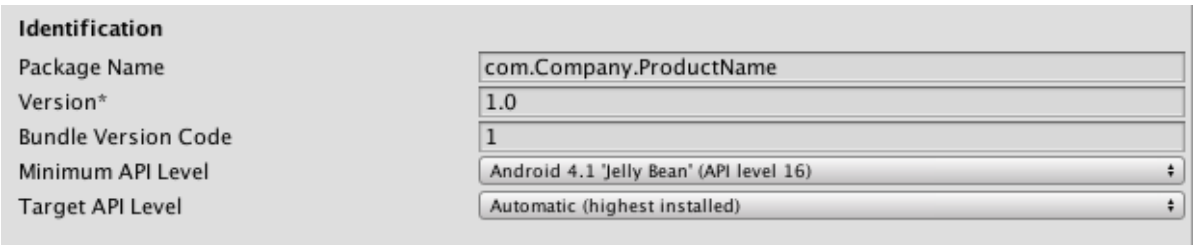
Property:	*Function:*
Color Space	The Color Space used for your application. See Linear Lighting for more information.
Auto Graphics API	Check this to make Unity auto-select a graphics API from the Open Graphics Library (OpenGL). When checked, Unity will try GLES3.1, and fall back to GLES3 or GLES2 if the device does not support GLES3.1. When unchecked, you can manually pick and reorder the graphics APIs. If only GLES3 is in the list, two additional checkboxes; **Require ES3.1** and **Require ES3.1+AEP** appear, allowing you to force the corresponding graphics API. *Important*: GLES3/GLES3.1/AEP requirement is added to your Android manifest only if GLES2 is not in the list, *and* Minimum API Level is set to JellyBean (API level 18) or higher. In this case only, your application won't show up on unsupported devices in Google Play Store.
Multithreaded Rendering	Check this box to move graphics API calls from Unity's main thread to a separate worker thread. This can help to improve performance in applications that have high CPU usage on the main thread.
Static Batching	Check this box to use Static batching on your build (enabled by default).
Dynamic Batching	Check this box to use Dynamic Batching on your build (enabled by default).
GPU Skinning	Check this box to enable OpenGL ES 3 GPU skinning. To learn more about GPU skinning, see the Wikipedia page on skeletal animation. Note that this property only supports VR apps, and only works if the Virtual Reality Supported checkbox is ticked.
Graphics Jobs (Experimental)	Check this box to instruct Unity to offload graphics tasks (render loops) to worker threads running on other CPU cores. This is intended to reduce the time spent in camera.render on the main thread. This feature is experimental. it may not deliver a performance improvement for your project, and may cause instabilities. Graphics Jobs are currently only supported when using Vulkan and the checkbox has no effect when using OpenGL ES.
Virtual Reality Supported	Enable this if your application is a virtual reality application, then add the required VR SDKs to the list.
Protect Graphics Memory	Check this box to force the graphics buffer to be displayed only through a hardware-protected path. Works only on devices which support it.

Vulkan Settings

The Vulkan Settings section of the Android PlayerSettings window

Property:	*Function:*
Enable SetSRGBWrite()	Use SRGB write mode during the frame. This may decrease performance, especially on tiled GPUs.
Use SW Commandbuffers	Build the command buffers in the task executor thread instead of building secondary command buffers for rendering API calls.

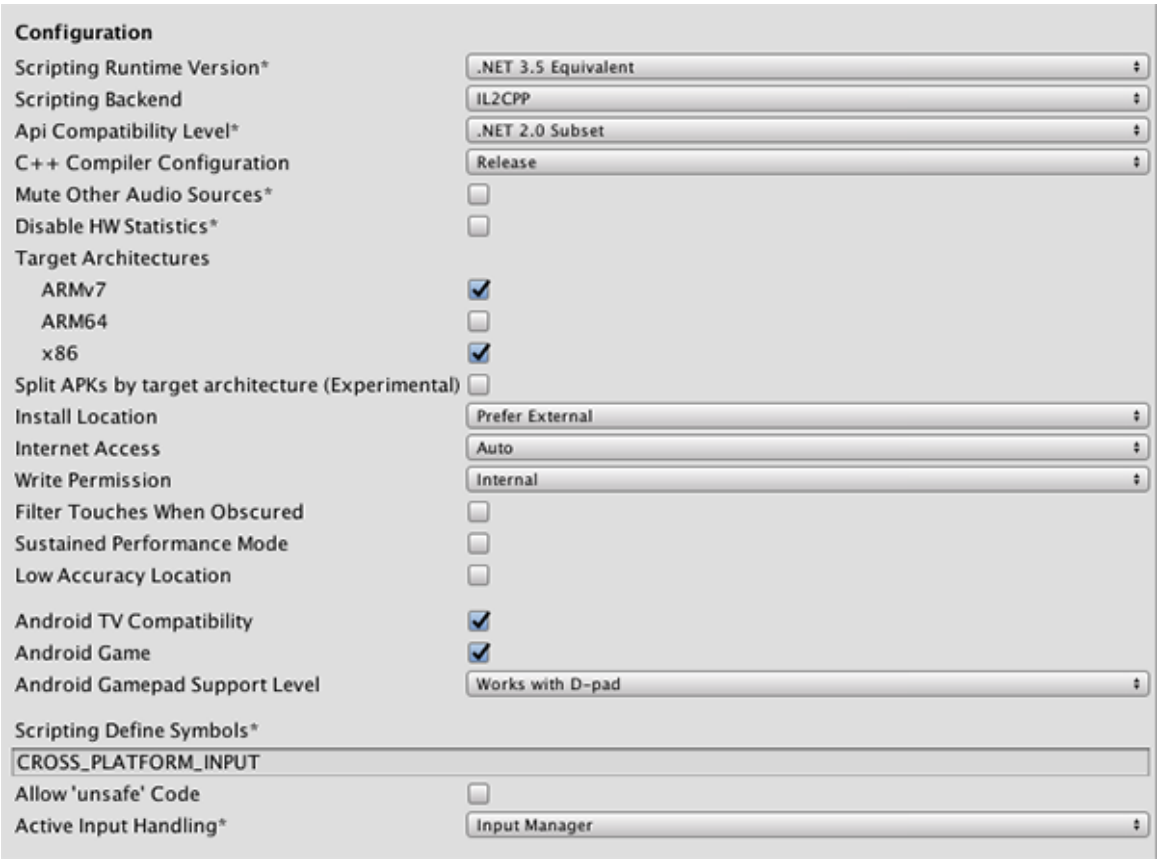
Identification



The Identification section of the Android PlayerSettings window

Property:	*Function:*
Package Name	The unique application ID, used to uniquely identify your app on the device and in Google Play Store. (Shared between iOS and Android.)
Version	Specifies the build version number of the bundle, which identifies an iteration (released or unreleased) of the bundle. The version is specified in the common format of a string containing numbers separated by dots (eg, 4.3.2). (Shared between iOS and Android.)
Bundle Version Code	An internal version number. This number is used only to determine whether one version is more recent than another, with higher numbers indicating more recent versions. This is not the version number shown to users; that number is set by the `versionName` attribute. The value must be set as an integer, such as “100”. You can define it however you want, as long as each successive version has a higher number. For example, it could be a build number. Or you could translate a version number in “x.y” format to an integer by encoding the “x” and “y” separately in the lower and upper 16 bits. Or you could simply increase the number by one each time a new version is released. Keep this number under 100000 if **Split APKs by target architecture** is enabled. Each APK must have a unique version code so Unity adds 100000 to the number for ARMv7, 200000 for ARM64 and 300000 for x86.
Minimum API Level	Minimum Android version (API level) required to run the application.
Target API Level	Target Android version (API level) against which to compile the application.

Configuration

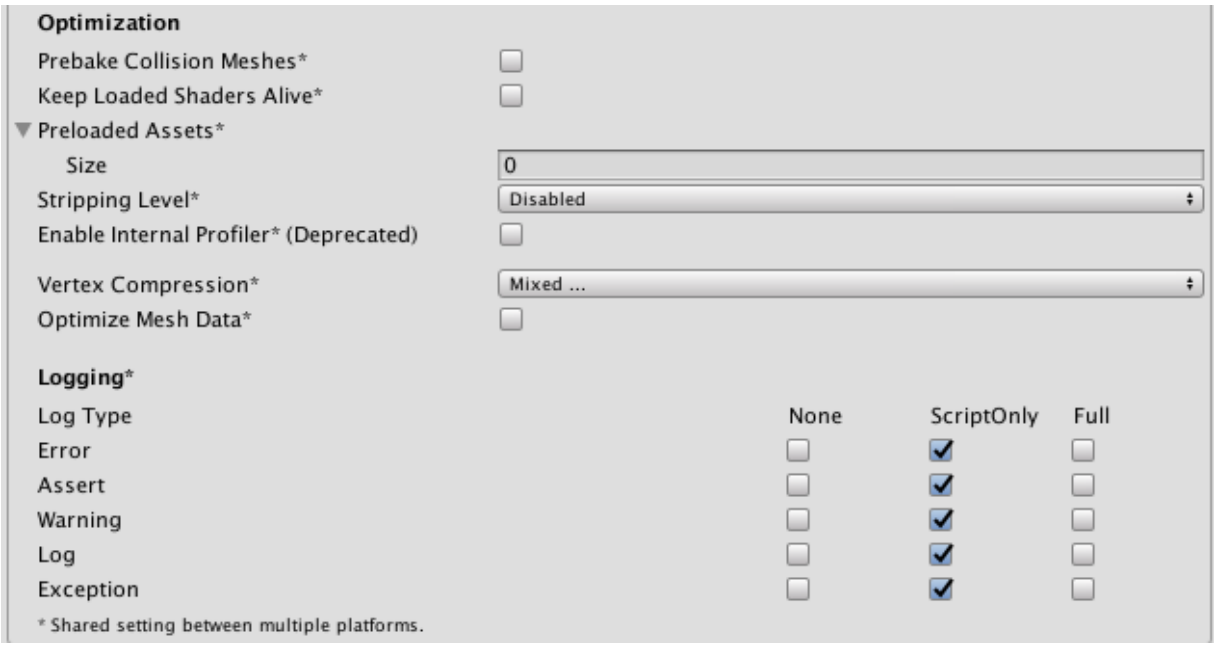


The Configuration section of the Android PlayerSettings window

Property:	*Function:*
Scripting Runtime Version	Choose which .NET runtime to use in your project. For more details, see Microsoft's .NET documentation.
.NET 3.5 Equivalent	A .NET runtime which implements the .NET 3.5 API. This functionality is deprecated, and should no longer be used. Please use .NET 4.
.NET 4.x Equivalent	A .NET runtime which implements the .NET 4 API. This API is newer than .NET 3.5, and as such, it offers access to more APIs, is compatible with more external libraries, and supports C# 6. This is the default scripting runtime.
Scripting Backend	Select your scripting backend. There are two options: IL2CPP or Mono2x. The default is Mono2x.
IL2CPP	Choose this to set IL2CPP as your scripting backend. See Unity documentation on IL2CPP for more information.
Mono	Choose this to set Mono as your scripting backend. This is the default option. See Mono's documentation for more information.
API Compatibility Level	Specifies active .NET API profile. See below.
	.Net 2.0: .NET 2.0 libraries. Maximum .NET compatibility, biggest file sizes
	.Net 2.0 Subset: Subset of full .NET compatibility, smaller file sizes
Mute Other Audio Sources)	Stops or allows audio from other applications to play in the background while your Unity application is running. Enable this and your Unity application stops audio from background applications, disable this and audio from background applications continue to play alongside your Unity application.

Property:	*Function:*
Disable HW Statistics	By default, Unity Android applications send anonymous HW statistics to Unity. This provides you with aggregated information to help you make decisions as a developer. Find these statistics at http://stats.unity3d.com/. Check this option to stop Unity sending these statistics.
Target Architectures	Allows the application to run on the specified CPUs. **Note**: Running Android apps in a 64-bit environment has performance benefits and 64-bit apps can address more than 4 GB of memory space.
Split APKs by target architecture	Create a separate **APK** for each the CPU architectures selected in **Target Architectures**. This makes download size smaller for Google Play Store users. This is primarily a Google Play store feature and may not work in other stores. For more details, refer to Multiple APK Support.
Install Location	Specifies application install location on the device (for detailed information, refer to Android Developer documentation on install locations.
Automatic	Let the operating system decide. User will be able to move the app back and forth.
Prefer External	Install the application to external storage (SD card) if possible. The operating system does not guarantee it; if not possible, the app will be installed to internal memory.
Force Internal	Force the application to be installed to internal memory. The user will be unable to move the app to external storage.
Internet Access	When set to **Require**, this adds the **networking** (`INTERNET`) permission to the Android manifest, even if you are not using any networking APIs. This is enabled by default for **development builds**.
Write Access	When set to **External(SDCard)** , enables write access to the external storage such as the SD card and adds a corresponding permission to the Android manifest. Enabled by default for development builds.
Sustained Performance Mode	Set a predictable and consistent level of device performance over longer periods of time, without thermal throttling. Overall performance might be lower when this setting is enabled. Based on the Android Sustained Performance API.
Android TV Compatibility	Check this box to mark the application as Android TV compatible.
Android Game	Android TV - Check this box to mark the output package (APK) will as a game rather than a regular application.
Android Gamepad Support Level	Android TV - This option allows you to define the level of support your application offers for a gamepad. The options are **Works with D-Pad**, **Supports Gamepad** and **Requires Gamepad**.
Scripting Define Symbols	Custom compilation flags (see the platform dependent compilation page for details).

Optimization



The Optimization section of the Android PlayerSettings window

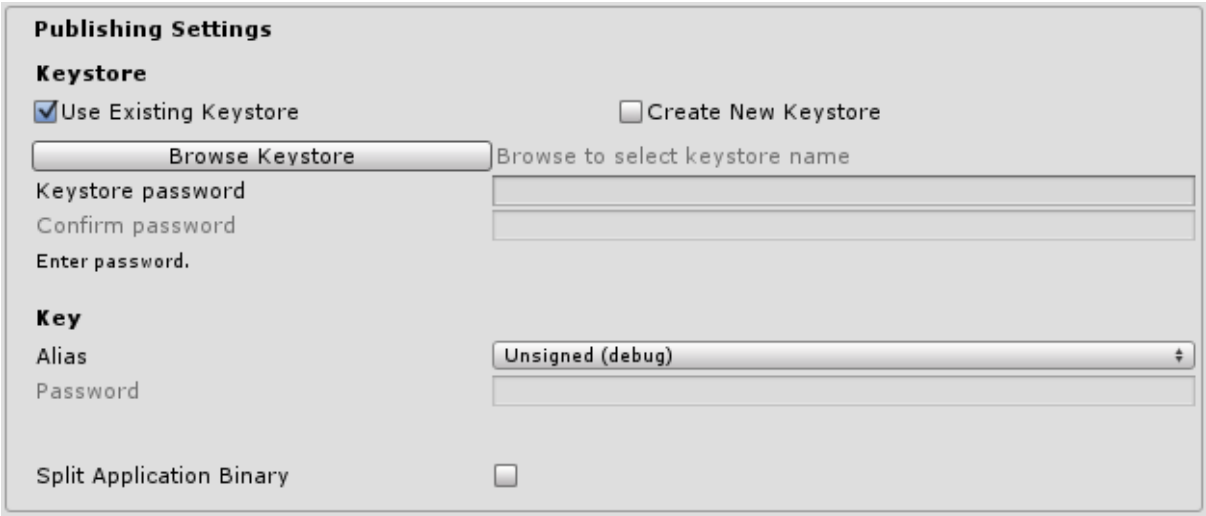
Property:	*Function:*
Prebake Collision__ Meshes||**Check this box to add** collision__A collision occurs when the physics engine detects that the colliders of two GameObjects make contact or overlap, when at least one has a rigidbody component and is in motion. More info See in Glossary data to meshes at build time.	
Keep Loaded Shaders__ Alive||**Check this box to prevent** shaders__A small script that contains the mathematical calculations and algorithms for calculating the Color of each pixel rendered, based on the lighting input and the Material configuration. More info See in Glossary from being unloaded.	
Preloaded Assets	An array of **assets** to be loaded when the application starts up.
Stripping Level	Options to strip out scripting features to reduce the output package (APK) size (This setting is shared between iOS and Android platforms, and is available with the Mono scripting backend only.)
	Disabled — No reduction is done.
	Strip Assemblies — Level 1 size reduction.
	Strip ByteCode (iOS only) — Level 2 size reduction (includes reductions from Level 1).
	Use micro mscorlib — Level 3 size reduction (includes reductions from Levels 1 and 2).

Property:	*Function:*
Strip Engine Code	Enable code stripping. (This setting is only available with the IL2CPP scripting backend.)
Enable Internal profiler	Check this box if you want to get the **profiler** data from your device in the Android SDK's adblogcat output while testing your projects (only available in development builds).
Vertex Compression	Select which vertex channels should be compressed. **Compression** can save memory and bandwidth but precision will be lower.
Optimize Mesh__ Data__	Remove any data from meshes that is not required by the material applied to them (tangents, normals, colors, UV).
Logging	Enable different logging types (see the StackTraceLogType page for details).

API Compatibility Level

You can choose your Mono API compatibility level for all targets. Sometimes a 3rd party .NET DLL uses the APIs that are not included into the .NET compatibility level you would like to use. In order to investigate such cases, get a reflection tool like ILSpy on Windows, and check the dependencies of the DLLs.

Publishing Settings



Property:	*Function:*
Keystore	
Use Existing Keystore / Create New Keystore	Use this to choose whether to create a new Keystore or use an existing one. You can use the **Browse Keystore** button to select a Keystore from the filesystem.

Property:	*Function:*
Keystore password	Create a password for the Keystore.
Confirm password	Enter the same password you did in **Keystore password** (only enabled if the Create New Keystore option is selected).
Key	
Alias	Key alias.
Password	Password for key alias.
Split Application Binary	Enable this option to split your output package into main (APK) and expansion (OBB) packages. This is required to publish applications larger than 100 MBytes to Goolge Play Store.

Note that for security reasons, Unity does not save the **Keystore** password or the **Key** password. The unsigned debug keystore is located by default at ~/.android/debug.keystore on MacOS and %USERPROFILE%\.android\debug.keystore on Windows.

XR settings

Property:	*Function:*
XR Settings	
Virtual Reality Supported	Enable native VR support for the Unity Editor and your game builds.
ARCore	Enable use of Google's ARCore platform.
XR Support Installers	
Vuforia Augmented Reality Supported	Enable use of the Vuforia Software Development Kit. You must have a Vuforia Software License and agree to the terms of that license before the property becomes enabled.

Details

Package Name

The **Package Name** string is the unique name of your application when published to Google Play Store and installed on the device. The basic structure of the identifier is **com.CompanyName.GameName**, and can be chosen arbitrarily. In Unity this field is shared with the **iOS** Player Settings for convenience.

Stripping Level

Most games don't use all the functionality of the DLLs included by default. With this option, you can strip out unused parts to reduce the size of the output package (APK).

2018–06–06 Page amended with limited editorial review

Updated features in 5.5

Sustained Performance Mode added in 2017.3

.NET 4.x runtime added in 2018.1

Android **Multiple APK** features added in 2018.2

Android 2D Textures Overrides

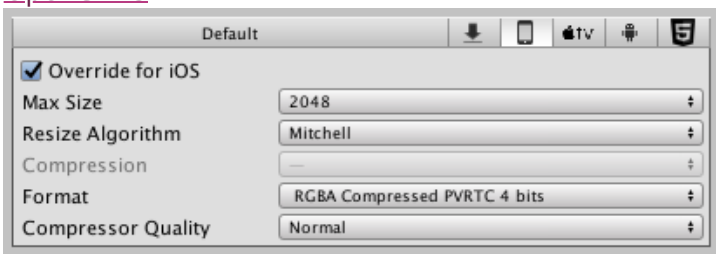
Leave feedback

This page details the **Texture Overrides** specific to Android. A description of the general Texture Overrides can be found here.

The information contained on this page assumes the reader has working knowledge of DXT and **ETC** Texture **compression**, OpenGL ES and Android development.

Further information about **Texture compression** and OpenGL ES can be found on the following Wikipedia pages:

DXT Texture compression
ETC(Ericsson Texture Compression
Lossy compression
OpenGL ES



Android 2D Texture Override settings

Texture Format	Internal representation
RGB Compressed DXT1	Compressed RGB texture. Supported by NVIDIA Tegra. 4 bits per **pixel** (32 KB for a 256x256 texture).
RGB Crunched DXT1	Compressed RGB Texture. Supported by NVIDIA Tegra. Crunch is a lossy compression format on top of DXT Texture compression. Textures are decompressed to DXT1 by the CPU and then uploaded to the GPU at runtime. Crunch produces smaller Textures than by regular DXT1 compression, but with lower quality. Crunch Textures can take a long time to compress, but decompression at runtime is very fast. 4 bits per pixel (output size varies depending on the Texture - from 1 KB for a 256x256 Texture).
RGBA Compressed DXT5	Compressed RGBA texture. Supported by NVIDIA Tegra. 8 bits per pixel (64 KB for a 256x256 texture).
RGBA Crunched DXT5	Compressed RGBA Texture. Supported by NVIDIA Tegra. Crunch is a lossy compression format on top of DXT Texture compression. Textures are decompressed to DXT5 on the CPU and then uploaded to the GPU at runtime. Crunch compression can be used to produce significantly smaller Textures than by regular DXT5 compression, but with lower quality. Crunch Textures can take a long time to compress, but decompression at runtime is very fast. 8 bits per pixel (output size varies depending on the Texture - from 1 KB for a 256x256 Texture).
RGB Compressed ETC 4 bits	Compressed RGB Texture. This is the default Texture format for Android projects in Unity. ETC_RGB4 is a part of OpenGL ES 2.0 and is supported by all OpenGL ES 2.0 GPUs. It does not support alpha. 4 bits per pixel (32 KB for a 256x256 Texture).
RGB Crunched ETC	Compressed RGB Texture. Crunch is a lossy compression format on top of ETC Texture compression. Textures are decompressed to ETC_RGB4 by the CPU and then uploaded to the GPU at runtime. Crunch produces smaller Textures than by regular ETC_RGB4 compression, but with lower quality. Crunch Textures can take a long time to compress, but decompression at runtime is very fast. ETC_RGB4 is a part of OpenGL ES 2.0 and is supported by all OpenGL ES 2.0 GPUs. It does not support alpha. 4 bits per pixel (output size varies depending on the Texture - from 1 KB for a 256x256 Texture).

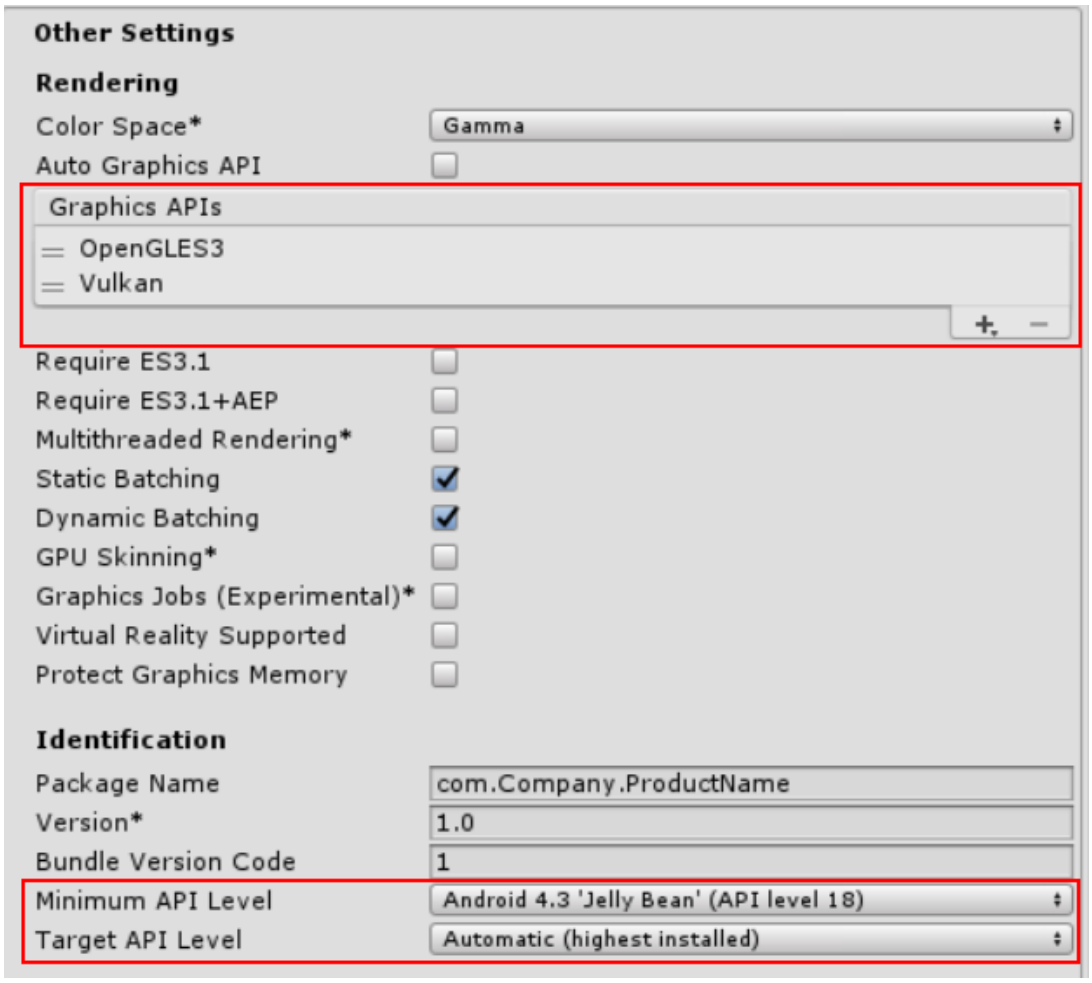
Texture Format	Internal representation
RGB Compressed ETC2 4 bits	Compressed RGB Texture. ETC2 is a part of OpenGL ES 3.0 and is supported by all OpenGL ES 3.0 GPUs. 4 bits per pixel (32 KB for a 256x256 Texture).
RGB + 1-bit Alpha Compressed ETC2 4 bits	Compressed RGB Texture with 1-bit punchthrough alpha. ETC2 is a part of OpenGL ES 3.0 and is supported by all OpenGL ES 3.0 GPUs. 4 bits per pixel (32 KB for a 256x256 Texture).
RGBA Compressed ETC2 8 bits	Compressed RGBA Texture. Supported by all OpenGL ES 3.0 GPUs. (64 KB for a 256x256 Texture)
RGBA Crunched ETC2	Compressed RGBA Texture. Crunch is a lossy compression format on top of ETC Texture compression. Textures are decompressed to ETC2_RGBA8 on the CPU and then uploaded to the GPU at runtime. Crunch compression can be used to produce significantly smaller Textures than by regular ETC2_RGBA8 compression, but with lower quality. Crunch Textures can take a long time to compress, but decompression at runtime is very fast. Supported by all OpenGL ES 3.0 GPUs. 8 bits per pixel (output size varies depending on the Texture - from 1 KB for a 256x256 Texture).
RGB Compressed PVRTC__ 2 bits__	Compressed RGB Texture. Supported by Imagination PowerVR GPUs. 2 bits per pixel (16 KB for a 256x256 Texture)
RGBA Compressed PVRTC 2 bits	Compressed RGBA Texture. Supported by Imagination PowerVR GPUs. 2 bits per pixel (16 KB for a 256x256 Texture)
RGB Compressed PVRTC 4 bits	Compressed RGB Texture. Supported by Imagination PowerVR GPUs. 4 bits per pixel (32 KB for a 256x256 Texture)
RGBA Compressed PVRTC 4 bits	Compressed RGBA Texture. Supported by Imagination PowerVR GPUs. 4 bits per pixel (32 KB for a 256x256 Texture)
RGB Compressed ATC__ 4 bits__	Compressed RGB Texture. Supported by Qualcomm Snapdragon. 4 bits per pixel (32 KB for a 256x256 Texture).
RGBA Compressed ATC 8 bits	Compressed RGBA Texture. Supported by Qualcomm Snapdragon. 8 bits per pixel (64 KB for a 256x256 Texture).

Texture Format	Internal representation
RGB Compressed ASTC 4x4 block	
RGB Compressed ASTC 5x5 block	
RGB Compressed ASTC 6x6 block	Compressed RGB Texture. Supported by all OpenGL ES 3.2 and OpenGL ES 3.1+AEP GPUs, as well as by some OpenGL ES 3.0 GPUs. This compression type uses fixed 128-bit block size, and depending on the pixel block size (4x4 to 12x12), it may utilize 8 to 0.89 bits per pixel. The size of the compressed Texture varies from 64 KB for a 256x256 Texture (4x4 block, highest quality) to 7.6 KB for a 256x256 Texture (12x12 block, highest compression rate).
RGB Compressed ASTC 8x8 block	
RGB Compressed ASTC 10x10 block	
RGB Compressed ASTC 12x12 block	

Texture Format	Internal representation
RGBA Compressed ASTC 4x4 block	
RGBA Compressed ASTC 5x5 block	
RGBA Compressed ASTC 6x6 block	Compressed RGBA Texture. Supported by all OpenGL ES 3.2 and OpenGL ES 3.1+AEP GPUs, as well as by some OpenGL ES 3.0 GPUs. This compression type uses fixed 128-bit block size, and depending on the pixel block size (4x4 to 12x12), it may utilize 8 to 0.89 bits per pixel. The size of the compressed Texture varies from 64 KB for a 256x256 Texture (4x4 block, highest quality) to 7.6 KB for a 256x256 Texture (12x12 block, highest compression rate).
RGBA Compressed ASTC 8x8 block	
RGBA Compressed ASTC 10x10 block	
RGBA Compressed ASTC 12x12 block	
RGB 16 bit	65 thousand colors with no alpha. Uses more memory than the compressed formats, but could be more suitable for **UI** or crisp Textures without gradients. 128 KB for a 256x256 Texture.
RGB 24 bit	Truecolor but without alpha. 192 KB for a 256x256 Texture.
Alpha 8 bit	High quality alpha channel but without any color. 64 KB for a 256x256 Texture.
RGBA 16 bit	Low-quality truecolor. The default compression for the Textures with alpha channel. 128 KB for a 256x256 Texture.
RGBA 32 bit	Truecolor with alpha - this is the highest quality compression for the Textures with alpha. 256 KB for a 256x256 Texture.
Compression quality	Choose Fast for quickest performance, Best for the best image quality and Normal for a balance between the two.

If a your app is installed on a device that does not support the Texture compression format you have used, Unity will decompress the Textures to RGBA 32 and store them in the device's memory—along with the compressed Textures—when the app is running. This will lead to increased loading times and higher memory usage as Unity must decompress the Textures and store two versions of the Textures used (compressed and uncompressed).This can also have a large impact on **rendering** performance, especially on older, slower devices.

Use ETC2 compression unless you are targeting specific hardware (like Nvidia Tegra). ETC2 supports Textures with and without an alpha channel and is supported by all OpenGL ES 3 devices. To avoid software decompression, set the minimum graphics API to OpenGL ES 3 in the **Player Settings** window by removing OpenGL ES 2 from the graphics API list and setting the Minimum API Level to 18+.



To target OpenGL ES 2 and OpenGL ES 3 devices, you can create two different APKs by first building an **APK** targeting OpenGL ES 3, as shown above, and then building an **APK** targeting OpenGL ES 2 by removing OpenGL ES 2 and Vulcan from the Graphics API section of the PlayerSettings window. You can then publish both of these APKs to the Google Play Store. When a user downloads your app, the Google Play Store will automatically install the most relevant **APK** for their device. For more information about publishing multiple **APK** files, see Android Developer documentation on Publishing Multiple APKs and Google Play Store APK filtering.

When building an APK that targets OpenGL ES 2, you can use ETC1 texture compression.

The Texture compression format for OpenGL ES2-targeted APK can be ETC1. Unity can use ETC1 for Textures with Alpha, provided they are placed on an Atlas (by specifying the packing tag) and the build is for Android. You can opt in for this by setting the "Compress using ETC1" checkbox for the Texture. Under the hood Unity will split the resulting atlas into two Textures, each without alpha and then combine them in the final parts of the render-pipeline.

If you absolutely want to store an alpha channel in a Texture, RGBA16 bit is the compression format supported by all hardware vendors.

2017–09–18 Page amended with editorial review

Updated feature in Unity 2017.1

Crunch compression format updated in Unity 2017.3

Gradle for Android

Leave feedback

Gradle is an Android build system which automates a number of build processes. This automation means that many common build errors are less likely to occur. Most notably in Unity, it reduces the method reference count in DEX (Dalvik Executable format) files, meaning that you are less likely to come across DEX limit problems. However, due to the differences between **Gradle** and the default Unity Android build system, some existing projects may be hard to convert to **Gradle**.

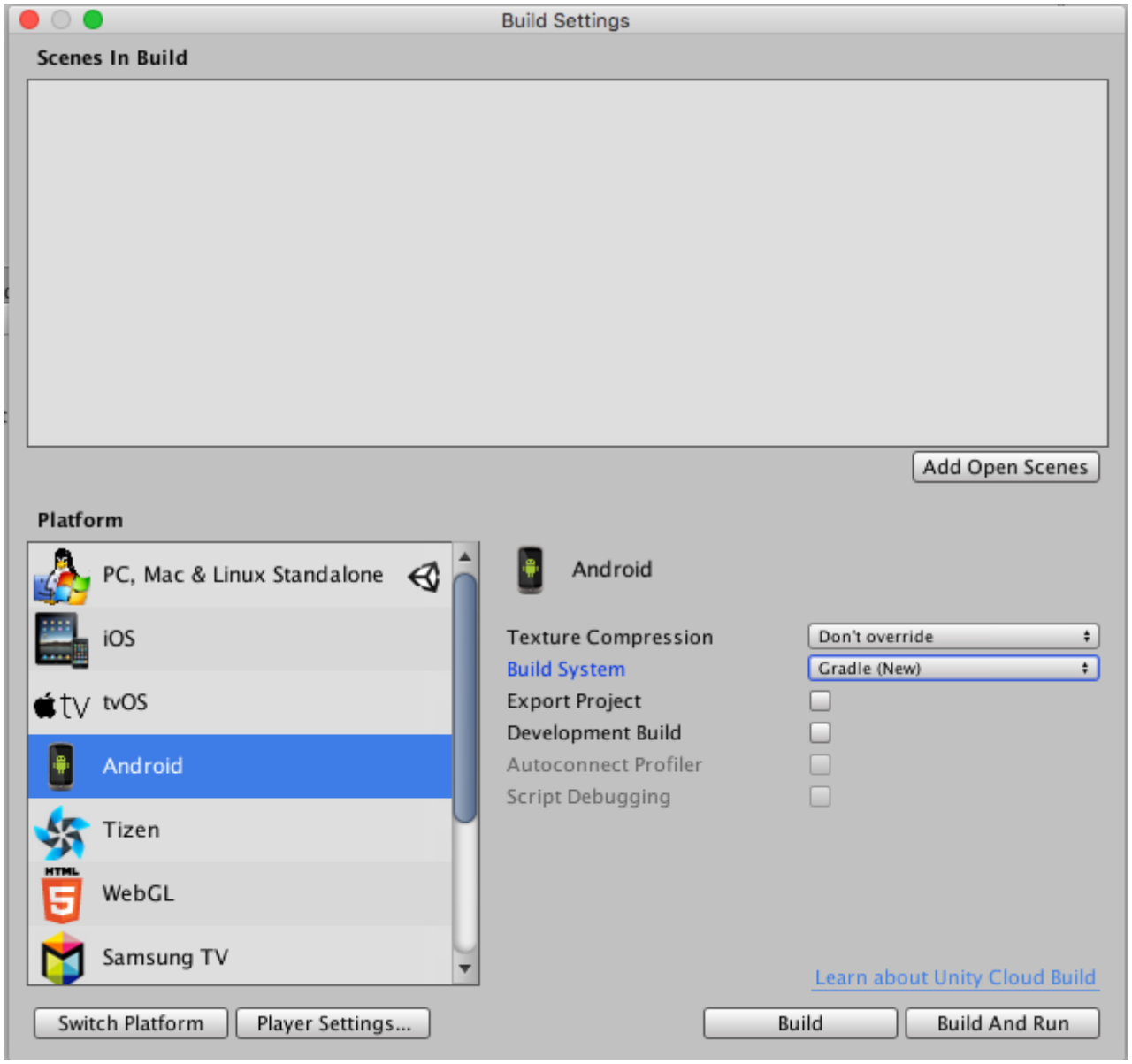
You can either build the output package (APK) using the Gradle build system in Unity, or export the Gradle project and build it in an external tool (such as Android Studio).

To learn more, see Gradle's resources on Getting Started with Gradle for Android Build.

Building with Gradle for Android

To build your Android build with Gradle in Unity:

In the Unity Editor, open the Build Settings window (menu: **File** > **Build Settings...**)
In the **Platform** list, select **Android**
Set the Build System drop-down to **Gradle (new)**, then click **Build**



Gradle build settings

Exporting the Gradle project

To export a Gradle project, follow the instructions above, but tick the **Export Project** option in the Build window before you click **Build**. When you click **Build** Unity generates a Gradle project in the specified directory rather than building the **APK**. Import this project into Android Studio to make additional modifications or to get full control of the build process.

See Android Studio's documentation on configuring your build for more information about building an output package (APK).

Providing a custom build.gradle template

To use your own build.gradle file when building the APK from Unity, use the Custom Gradle Template checkbox under Player Settings. This will generate a default mainTemplate.gradle for you, which you can edit. This template contains several variables that will be filled in by the Unity build process, such as `**TARGETSDKVERSION**` – you would normally leave these alone. You can also use your own settings.gradle by providing a settingsTemplate.gradle in your **Plugins** directory in the same way (although there is currently no automatic way

to do this). This file is responsible for including your library projects, so unless you want to also override that process, the file should at least contain this line:

```
**INCLUDES**
```

which will be replaced by include directives to all your libraries.

Template Variables

These variables can be used in the mainTemplate.gradle file;

Variable:	Description:
DEPS	List of project dependencies; ie the libraries used.
API VERSION	API version to build for (eg 25).
BUILDTOOLS	SDK Build tools used (eg 25.0.1).
TARGETSDKVERSION	API version to taget (eg 25).
APPLICATIONID	Android Application ID (eg com.mycompany.mygame).
MINIFY_DEBUG	Minify for debug builds enabled (true or false).
PROGUARD_DEBUG	Use proguard for minification (true or false.)
MINIFY_RELEASE	Minify for release builds enabled (true or false).
PROGUARD_RELEASE	Use proguard for minification (true or false).
USER_PROGUARD	Custom user proguard file (ie progard-user.txt).
SIGN	Complete signingConfigs section if build is to be signed.
SIGN_CONFIG	Set to 'signingConfig signingConfig.release' if build is signed.
DIR_GRADLEPROJECT	Directory where the gradle project is created.
DIR_UNITYPROJECT	Directory of your Unity project.

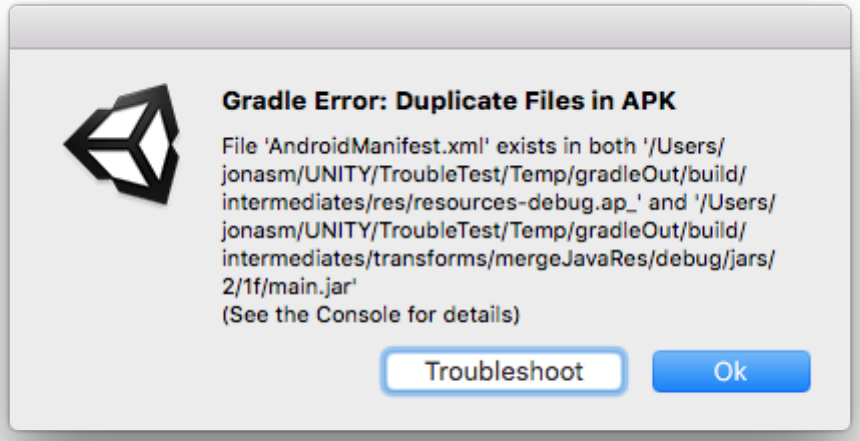
Minification

You can activate **Proguard Minification** under Player Settings Minify. Note that proguard can easily strip out code that is actually needed, so often this process needs to be configured carefully. You can generate a custom proguard.txt using the checkbox User Proguard File under the same settings as a starting point.

See the ProGuard manual for more information on Proguard.

Errors when building with Gradle

If an error occurs during building for Android using Gradle, Unity displays an error dialog box. Click **Troubleshoot** to open the Gradle troubleshooting Unity documentation in your system's browser.



Unity's Grade build error dialog box

2017–10–02 Page published with limited editorial review

Expanded custom build.gradle template section.

New feature in 5.5

Gradle troubleshooting

Leave feedback

If you just switched to export your Android project using **Gradle** instead of the old system, you may encounter build errors, especially if you are using additional Android libraries, or if you have added a custom **AndroidManifest.xml**.

The Android Gradle plug-in is much more picky than the old **ADT**/Ant system. It does not accept anything it considers an error, whether it's duplicate symbols, references to resources that don't exist, or a library project that sets the same attribute as the main application.

In most cases, fixing the problem involves editing an **AndroidManifest.xml** file; either the main one, or one from a library your project uses.

In a non-trivial project, or if the project has issues not described by the troubleshooting section below, export the project as a Gradle project (from **Build Settings**) and build from the command line. Building from the command line gives you more detailed error messages, and makes for a quicker turnaround when applying changes.

Specific problems

Resource not found

An **AndroidManifest.xml** file, either the main one or in a library, references a non-existing resource. Often it is the application icon or label string that is set by a library. This can happen if you have copied your main manifest to a library project without removing those references.

Remove the attribute from one of the Android Manifests – normally the one from the library.

Duplicate files in APK

You have a file name **collision** between your main application and a library project, or between two library projects. Keep in mind that all of the files are copied into the same **APK** package.

You need to remove one of the files.

Colliding package names

A library can not use the same Java package as the main application, or any other library.

Usually, you should change the package name of the library to something different. If the library contains a lot of code, it may be easier to change the main package name (from *Player Settings*).

Colliding attributes

A library can not freely override attributes from the main **AndroidManifest.xml**. Often this error is caused by a library setting the application icon or label string, similar to the **Resource not found** problem above.

Either remove the attribute from the library, or add a **tools:replace** attribute to your **application** tag, to indicate how the merge conflict should be resolved.

Android Manifest

Leave feedback

The Android Manifest is an XML file which contains important metadata about the Android app. This includes the package name, activity names, main activity (the entry point to the app), configurations, Android version support, hardware features support, and permissions.

For more information about the Android Manifest file, refer to the Android Developer documentation on Android Manifests.

How Unity produces the Android Manifest

When building your app, Unity automatically generates the Android Manifest, following the steps below:

Unity takes the main Android Manifest.

Unity finds all the Android Manifests of your plug-ins (AARs and Android Libraries).

Manifests from plug-ins are merged into the main Manifest using Google's manifmerger class.

Unity modifies the Manifest, automatically adding permissions, configuration options, features used, and other information to the Manifest.

Overriding the Android Manifest

Although Unity generates a correct Manifest for you, in some cases you might want direct control over its contents.

To use an Android Manifest that you have created outside of Unity, import your Android Manifest file to the following location: *Assets/**Plugins**/Android/AndroidManifest.xml*. This overrides the default Unity-created Manifest.

In this situation, Android Libraries' Manifests are later merged into your main Manifest, and the resulting Manifest is still tweaked by Unity to make sure the configuration is correct. For full control of the Manifest, including permissions, you need to export the Project and modify the final Manifest in Android Studio. Please note that we only support *launchMode - singleTask* .

Permissions

Unity automatically adds the necessary permissions to the Manifest based on the Player Settings and Unity APIs that your app calls from the script. For example:

Network classes add the `INTERNET` permission

Using vibration (such as Handheld.Vibrate) adds `VIBRATE`

The InternetReachability property adds `ACCESS_NETWORK_STATE`

Location APIs (such as LocationService) adds `ACCESS_FINE_LOCATION`

WebCamTexture APIs add `CAMERA` permission

The Microphone class adds `RECORD_AUDIO`

For more information about permissions, see the Android Manifest Permissions page of the Android Developer documentation.

Note that if your plug-ins require a permission by declaring it in their Manifests, the permission is automatically be added to the resulting Android Manifest during the merge stage. Any Unity APIs called by the plug-ins also contribute to the permission list.

Runtime permissions in Android 6.0 (Marshmallow)

If your app is running on a device with Android 6.0 (Marshmallow) or later and also targets Android API level 23 or higher, your app uses the Android Runtime Permission System.

The Android Runtime Permission System asks the app's user to grant permissions while the app is running, instead of when the app is first installed. App users can usually grant or deny each permission when the app needs them while the app is running (for example, requesting camera permission before taking a picture). This allows an app to run with limited functionality without permissions. The Android.Permission class allows you to check whether specific permissions have been granted or denied. If a permission your app relies upon has been denied, you can inform the user why you need it and ask them to approve the permission. See Requesting Permissions.

Normally, your app prompts the user to allow what Android calls "dangerous" permissions on startup. See Android's documentation on dangerous permissions for more information. However, if you don't want your Unity Android app to ask for permissions on startup, you can add the following to your Manifest in either the **Application** or **Activity** sections.

```
<meta-data android:name="unityplayer.SkipPermissionsDialog" android:value="true"
```

Adding this completely suppresses the permission dialog shown on startup, but you must handle runtime permissions carefully to avoid crashes. This method is only recommended for advanced Android app developers.

For further information about the Runtime Permission System and handling permissions, see the Requesting Permissions section of the Android Developer documentation.

Examining the resulting Android Manifest

To examine the final Android Manifest that Unity has generated for your app, open the *Temp/StagingArea/AndroidManifest.xml* file after you have built your Project but before exiting the Unity Editor.

The Manifest is stored in binary format in the output package (APK). To check the contents of a Manifest inside an **APK**, you can use the Android Studio APK Analyzer or another third-party tool (such as Apktool).

2018–10–05 Page published with editorial review

Updated functionality in 5.5

Added support for Android Runtime Permissions in 2018.3.

Requesting Permissions

Leave feedback

On Android 6 (API level 23) and above, the Android.Permission API allows you to request permission to use some commonly needed system features, such as the **camera**, microphone, or location, when they are needed rather than when the application starts up. You can request each individual permission when needed and show the user a message that explains the reason why you are asking.

The standard method Android uses for permissions is to show the user a list of the permissions that the app needs when it starts up, providing no explanation as to why or in what context these permissions are used. This can be confusing and some users might deny the permissions even if they are necessary to the operation of the application.

Google's guideline for requesting permissions recommends that, if the user has denied your permission request once, you should display the reason for the request and give them the opportunity to allow you to present the request again.

You can use the Permission.HasUserAuthorizedPermission function to check whether a permission has been authorized by the user. If it has not, you can request the permission again, explaining why your app needs the permission. Once you have the user's approval, call Permission.RequestUserPermission to request the permission again. When you call this function, Android opens the system permission dialog to allow the user to approve (or deny) the permission.

If the user still denies the permission request, you should disable the related functionality in your app, or, if the app cannot work without it, inform the user. Note that if the user previously checked the "Do not ask me again" option on the system permission dialog, `RequestUserPermission()` does not open the system dialog. The user must go into the application permission settings and manually turn on the permission in this case.

You can add the `unityplayer.SkipPermissionsDialog` meta-data to your Android manifest to suppress the permissions dialog normally shown when the app starts up. If you do, you must call Permission.RequestUserPermission at an appropriate time to gain access to each protected feature. See Android Manifest for more information.

For additional information on requesting Android permissions, see App permissions best practices in the Android developer guide.

The following sample code shows how to check whether a permission has been granted for a specific permission and present a dialog in the case where the user has denied the permission:

MicrophoneTest.cs

```
using UnityEngine;
#if PLATFORM_ANDROID
using UnityEngine.Android;
#endif

public class MicrophoneTest : MonoBehaviour
{
```

```
    GameObject dialog = null;

    void Start ()
    {
        #if PLATFORM_ANDROID
        if (!Permission.HasUserAuthorizedPermission(Permission.Microphone))
        {
            Permission.RequestUserPermission(Permission.Microphone);
            dialog = new GameObject();
            }
        #endif
    }

    void OnGUI ()
    {
        #if PLATFORM_ANDROID
        if (!Permission.HasUserAuthorizedPermission(Permission.Microphone))
        {
            // The user denied permission to use the microphone.
            // Display a message explaining why you need it with Yes/No buttons.
            // If the user says yes then present the request again
            // Display a dialog here.
            dialog.AddComponent<PermissionsRationaleDialog>();
            return;
        }
        else if (dialog != null)
        {
            Destroy(dialog);
        }
        #endif

        // Now you can do things with the microphone
    }
}
```

PermissionsRationaleDialog.cs

```
using UnityEngine;
#if PLATFORM_ANDROID
using UnityEngine.Android;
#endif

public class PermissionsRationaleDialog : MonoBehaviour
```

```
{
    const int kDialogWidth = 300;
    const int kDialogHeight = 100;
    private bool windowOpen = true;

    void DoMyWindow(int windowID)
    {
        GUI.Label(new Rect(10, 20, kDialogWidth - 20, kDialogHeight - 50), "Plea
        GUI.Button(new Rect(10, kDialogHeight - 30, 100, 20), "No");
        if (GUI.Button(new Rect(kDialogWidth - 110, kDialogHeight - 30, 100, 20)
        {
            #if PLATFORM_ANDROID
            Permission.RequestUserPermission(Permission.Microphone);
            #endif
            windowOpen = false;
        }
    }

    void OnGUI ()
    {
        if (windowOpen)
        {
            Rect rect = new Rect((Screen.width / 2) - (kDialogWidth / 2), (Scree
            GUI.ModalWindow(0, rect, DoMyWindow, "Permissions Request Dialog");
        }
    }
}
```

Note: Add `MicrophoneTest` to a **GameObject** as a **component**, but not `PermissionsRationaleDialog`. `MicrophoneTest` automatically creates a GameObject and adds `PermissionsRationaleDialog` when it needs to show the dialog. Also note that your app must use the Microphone class for Unity to add the microphone permission.

2018–10–03 Page published with editorial review
New feature in 2018.3

Facebook

Leave feedback

This section of the User Manual contains documentation on developing for the Facebook platform. Use the navigation column on the left-hand side of the page to view topics in this section.

Getting started with Facebook development

Leave feedback

What is the Facebook build target?

The Facebook build target makes it easy to publish Unity games to Facebook and to use Facebook functionality in your games. Using the Facebook build target, you can build your content either as a WebGL player, which you can then publish to facebook.com, or as a custom native Windows Standalone player, which you can the publish to the Facebook Gameroom client.

When the Facebook build target is selected, you automatically have access to the Facebook SDK in your **scripts**, which lets you interact with Facebook and access it's social features.

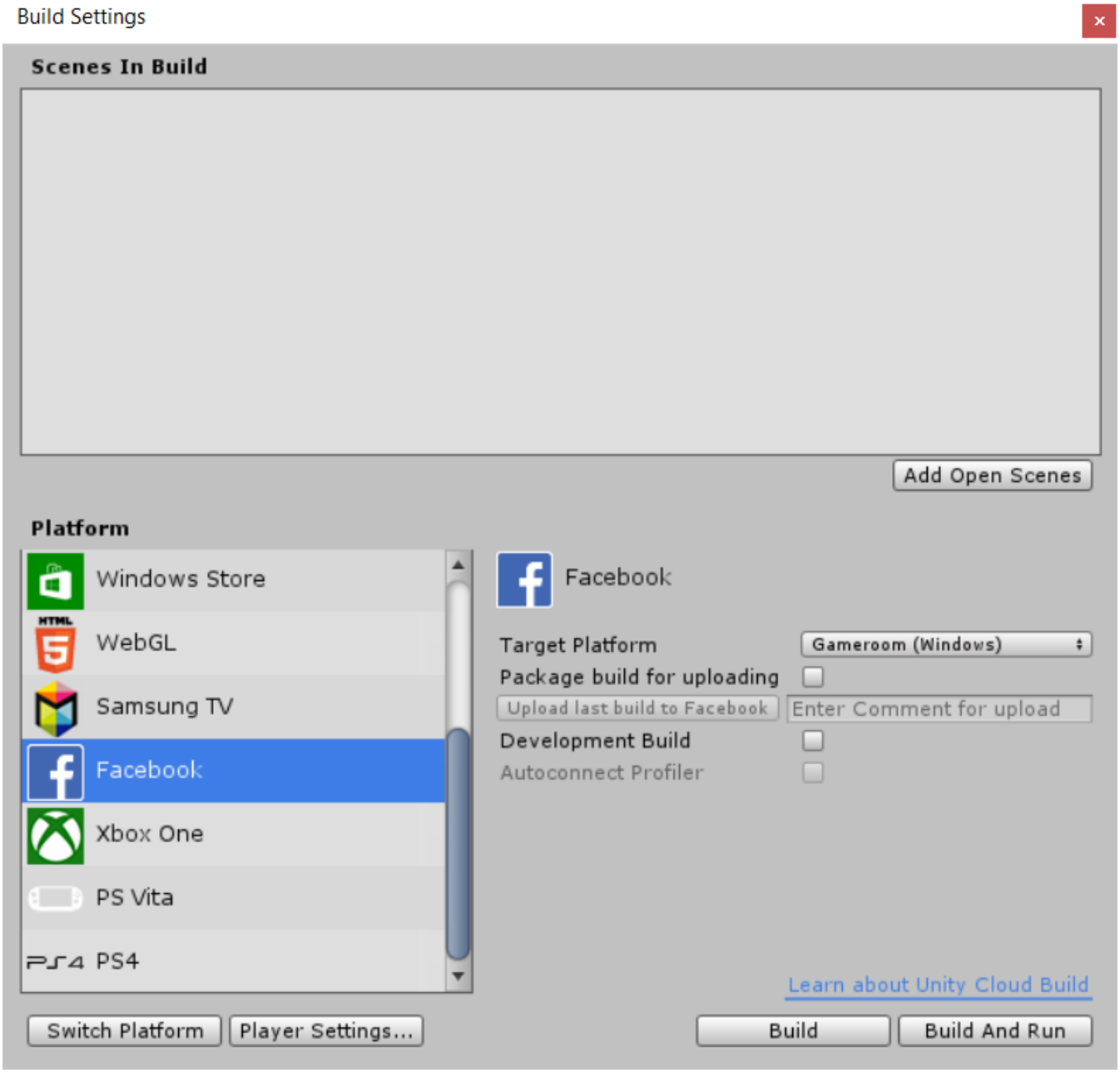
Publishing your game to Facebook

Configuration

To publish your game to Facebook, you first need to create a new App on the Facebook developer page. Once completed, this will give you an AppID, which you should paste into your Facebook PlayerSettings. Now, you can get an *Upload access token* from Facebook, on your app configuration page, under the **Web Hosting** tab. Also paste this into your Facebook Player Settings. This will allow you to upload your game to Facebook directly from the Unity Editor.

Building

Access the Facebook Build Settings via the in the **Build Settings** dialog box (menu: **File** > **Build Settings...**). In the dialog box, select **Facebook** from the **Platform** list.



Here you can choose to build your content as WebGL or as a Windows Standalone for Gameroom. If you plan to upload your game, choose **Package build for uploading**, which generates a compressed package, which can be uploaded to facebook.

After making a build, the **Upload last build to Facebook** button becomes available. If you have correctly configured your AppID and Upload access token, you can click this button to upload your build to facebook. The **Enter Comment for upload** field lets you specify an optional comment to identify your build.

Once you have uploaded a build to Facebook, it will appear on your your app configuration page on Facebook, under the **Web Hosting** tab. Here you can choose to push your build to production or to stage partial rollouts to smaller percentages of your users.

Using the Facebook SDK

When the Facebook build target is active, you can use the Facebook SDK in your scripts. This lets you share content on Facebook, track **analytics** events, use Facebook Payments and more. For more information on how to use the SDK, see Facebook's documentation.

Select which version of the SDK you want to use in the Unity Editor's Facebook Player Settings (click the **Player Settings** button in the Build Settings window, or select menu: **Edit > Project Settings > Player**). These settings show all the versions that Facebook has made available for your version of Unity.

Using a custom Facebook SDK

If you want to use a different build of the Facebook SDK then the one included by Unity by default, you can do that, as long as it at least version 7.9.1 (so it supports the Facebook build target). Just download a build of the SDK from Facebook, and drop it into your **Assets** Folder. Unity will detect this, and disable the built-in SDK. This will allow you to use the Facebook SDK outside of the Facebook build target, which you want to use Facebook features on other platforms facebook supports.

2017–05–16 Page published with no editorial review

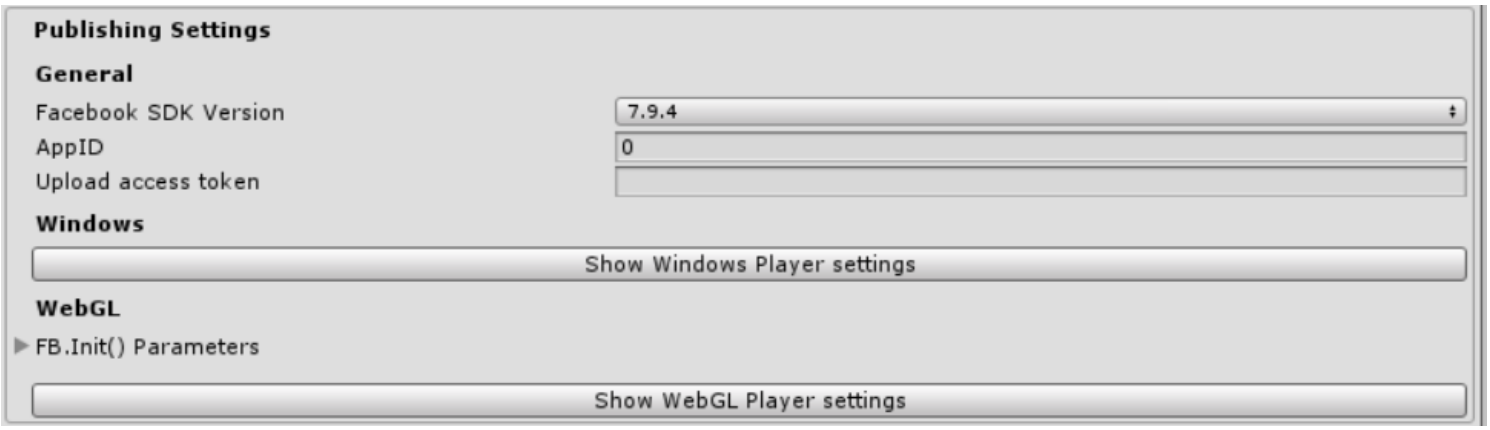
Facebook Player Settings

Leave feedback

This page details the **Player Settings** specific to the Facebook platform. For descriptions of general settings, see Player Settings and Splash Screen settings.

Note that the Facebook build target uses the existing WebGL and Windows Standalone build targets, so the Player Settings for those targets will also apply.

Publishing settings



Property	Function
Facebook SDK Version	Select the Facebook SDK Version for your project to use. This menu will be populated with new versions compatible with your version of Unity as Facebook publishes them.
AppID	Your Facebook AppID, used by Facebook to identify your app. See Facebook's getting started page on how to set one up.
Upload access token	An upload access token, required to authorize the Unity editor to upload builds of your app to Facebook. You can get this from Facebook, on your app configuration page, under the "Web Hosting" tab.
Show Windows Player Settings	Open the Standalone Player settings, which affect any Facebook builds for Gameroom.
Show WebGL Player Settings	Open the WebGL Player settings, which affect any Facebook builds for WebGL.
FB.Init() Parameters	Some parameters affecting how the Facebook SDK is initialized on the facebook.com web page. Documented here.

2017–05–16 Page published with no editorial review

Windows

Leave feedback

This section of the User Manual contains documentation on developing for the Windows platform. Use the navigation column on the left-hand side of the page to view topics in this section.

Windows General

Leave feedback

This section contains information which can be applied for several Windows platforms.

Windows standalone Player build binaries

Leave feedback

When you build a Unity project to the Windows standalone platform Unity produces the following files (where 'ProjectName' is the name of your project):

`ProjectName.exe` - The project executable. This contains the program entry point which calls into the Unity engine when launched.

`UnityPlayer.dll` - this DLL contains all the native Unity engine code. It is also signed using the Unity Technologies certificate, allowing easy verification that the engine was not tampered with.

`*.pdb files` - Symbol files for debugging. Unity copies these to the build directory if you enable **Copy PDB files** in the Build Settings window.

`WinPixEventRuntime.dll` - This DLL enables Windows PIX profiler support. Unity only creates this file if you check the **Development Build** checkbox in the Build Settings window.

`ProjectName_Data folder` - This folder contains all the data needed to run your project.

Rebuilding the game executable

Unity creates the source code for `ProjectName.exe` in this folder: *Editor\Data\PlaybackEngines\WindowsStandaloneSupport\Source\WindowsPlayer*.

To modify the executable, or ship code which you built yourself (if you want to sign it, for example), you must rebuild the executable and place it in your built game directory.

To build your executable outside of Unity, you need Visual Studio 2015 or 2017 with 'Common Tools for Visual C++" and Windows support for C++ installed.

2017–07–19 Page published with limited editorial review

Windows standalone Player build binaries changed in 2017.2

Windows XP support in Standalone Player removed in 2018.1

Windows Debugging

Leave feedback

Unity provides some facilities to ease the debugging on Windows for forensic or live debugging of game or editor processes.

Native vs. Managed Debugging

First, clarity regarding debugging. There are two types of debugging that need addressing within Unity. There is the native C++ debugging as well as the C# managed debugging. For platforms supporting **IL2CPP**, there will be only native debugging, but managed debugging will stay for the editor for fast iteration purposes.

Native Debugging

Native Debugging is facilitated by having symbols (pdb files) for the associated binary files (exe or dll).

Managed Debugging

On Windows, the standard .NET managed symbols are stored in pdb files as well, however when using mono, there are mdb files.

Symbols

Unity provides a symbol store at http://symbolserver.unity3d.com/ . This server URL can be utilized in windbg or VS2012 and later for automatic symbol resolution and downloading (much like Microsoft's symbol store).

Windbg Setup

The easy way to add a symbol store on windbg is the .sympath command.

```
.sympath+ SRV*c:\symbols-cache*http://symbolserver.unity3d.com/
```

Let's break that down:

`.sympath+`
The + addition, leaves the existing symbol path alone, and appends this symbol store lookup

`SRV*c:\symbols-cache`
The SRV indicates a remote server to fetch from, while the c:\symbols is a local path to cache the downloaded symbols and to look there first before downloading again.

`*http://symbolserver.unity3d.com/`
The path to the symbol store to fetch from

Visual Studio Setup

Note: VS2010 and earlier do not function with http server symbol stores.
1. Go to Tools -> Options
2. Expand the Debugging Section, select Symbols
3. Specify a cache directory (if not already specified)
4. Add a "Symbol file (.pdb) location" of http://symbolserver.unity3d.com/

Live Debugging

Live Debugging is the scenario of attaching a debugger to a process that is running normally, or to a process where an exception that has been caught. In order for the debugger to know what's going on, the symbols need to be included in the build. That's what the steps above should address. The one additional thing to know is that the game executable is

named according to your game name, so the debugger may have issues finding the correct pdb if it doesn't have access to your renamed executable.

Setting up automatic exception debugging

On Windows, Microsoft sets up automatically on application crashes to go to Dr Watson/Error Reporting to Microsoft. However, if you have Visual Studio or windbg installed, Microsoft provides a facility to savvy developers to instead opt to debug the crashes.
For ease of installing, here's a registry file contents to install:

```
Windows Registry Editor Version 5.00
[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AeDebug]
"Auto"="1"
[HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\Microsoft\Windows NT\CurrentVersion\AeDebug]
 "Auto"="1"
```

A little extra for editor debugging:

`Unity.exe -dbgbreak`
Will launch Unity and immediately offer a debugger to connect if the automatic crash handling is set up

Post-Mortem/Forensic Debugging

Windows provides facilities to investigate crash dump files (.dmp or .mdmp). Depending on the type of crash dump, there may simply be stack information or perhaps the entire process memory. Depending on the contents, various possibilities exist in seeing what may have happened to cause a crash. In the usual case, you often at least have a stack to investigate (if it's a valid stack...)

To investigate a dump file, your options are to load up via Visual Studio or windbg. While Visual Studio is a more friendly tool to use, its power is a bit more limited than windbg.

Debugging Hints and Tricks

Managed exceptions in native land

A NullReferenceException will often look like this:

```
    1b45558c()
>   mono.dll!malloc(unsigned int size=12)  Line 163 + 0x5f bytes    C
    mono.dll!g_hash_table_insert_replace(_GHashTable * hash=0x065c3960, void * key=0x
    mono.dll!mono_jit_runtime_invoke(_MonoMethod * method=0x242bf8b0, void * obj=0x06
```

This is not a crash in malloc, nor in mono - it's a NullReferenceException that's either:
* Caught by the VS debugger * Unhandled in a user's player, causing the player to exit

Viewing managed stack frames

With the previous example again:

```
    1b45558c()
>   mono.dll!malloc(unsigned int size=12)  Line 163 + 0x5f bytes    C
    mono.dll!g_hash_table_insert_replace(_GHashTable * hash=0x065c3960, void * key=0x
    mono.dll!mono_jit_runtime_invoke(_MonoMethod * method=0x242bf8b0, void * obj=0x06
```

The lines without any information are managed frames. There is, however, a way to get the managed stack information: mono has a builtin function called mono_pmip, which accepts the address of a stack frame and returns a char* with information. You can invoke mono_pmip in the Visual Studio immediate window:

```
?(char*)mono.dll!mono_pmip((void*)0x1b45558c)
```

0x26a296c0 " Tiles:OnPostRender () + 0x1e4 (1B4553A8 1B4555DC) [065C6BD0 - Unity Child Domain]"`

Note: This only works where mono.dll symbols are properly loaded.

Force Applications to Create Dumps

Sometimes there are cases where the application doesn't crash with the debugger attached, or an application crashes on a remote device where the debugger is not available. However, you can still get useful information if you can get the dump file - follow the below steps in order to do so.

Note: These instructions are for Windows Standalone and **Universal Windows Platform** (when running on desktop).

Open the registry.
Navigate to HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\Windows Error Reporting
Create LocalDumps folder if it's not there.
Add following keys:
"DumpFolder"=<FolderPath goes here> , for example, C:\Temp
"DumpCount"=dword:00000010
"DumpType"=dword:00000002
Launch the application (it can be anything that runs on Windows - Universal Windows Platform or Windows Standalone Executable)
Reproduce the crash, the dump file should be created in your specified folder. You can either open the dump file with Visual Studio or WinDbg.

• 2017–05–16 Page amended with no editorial review

WindowsLowIntegrity

Leave feedback

Windows Standalone player can detect if it's running at Low Integrity Level (see Microsoft's documentation on Designing Applications to Run at a Low Integrity Level for more information). In this case, several things happen:

The log file is written to %USER PROFILE%\AppData\LocalLow**CompanyName****ProductName**
PlayerPrefs is saved to HKCU\Software\AppDataLow\Software**CompanyName****ProductName**

Windows Player: IL2CPP Scripting Backend

Leave feedback

IL2CPP is a fully-supported **scripting backend** that can be used as an alternative to Mono when building projects for Windows Player.

When you build a Project using IL2CPP, Unity converts IL code from **scripts** and assemblies to C++ before creating a native binary. See IL2CPP for more information.

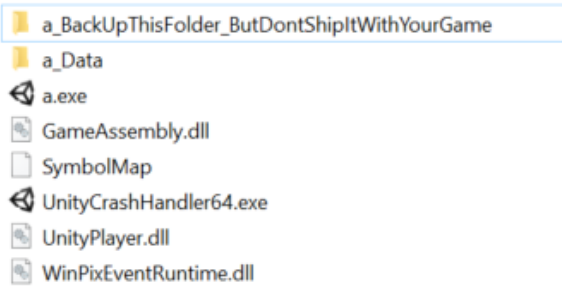
• 2018–03–13 Page published with editorial review

New feature in 2018.1

Windows Player: IL2CPP build files

Leave feedback

A Project using the **IL2CPP scripting backend** will typically (can differ due to build settings) produce these files:



IL2CPP files produced during build

The following files are common to Projects using either Mono or IL2CPP:

File:	Description:
a_Data	Folder with game data.
a.exe	Main game executable.
UnityCrashHandler64.exe	Crash handler executable.
UnityPlayer.dll	Unity Player library containing all native code.
WinPixEventRuntime.dll	PIX for Windows runtime. This file will only be present in **development builds**.

The following files are common to Projects using either Mono or IL2CPP:

File:	Description:
a_BackUpThisFolder_ButDontShipItWithYourGame	Folder containing data required to debug your game, including PDB (debug info) files and C++ code generated from your **scripts**. You should back up this folder for every build you ship, but don't redistribute it.
GameAssembly.dll	Library that contains the IL2CPP runtime and all your script code.
SymbolMap	File containing a list of all managed function addresses and their lengths. IL2CPP needs this to resolve managed stack traces. If you delete it, your game will still run but exceptions will not produce sensible call stacks

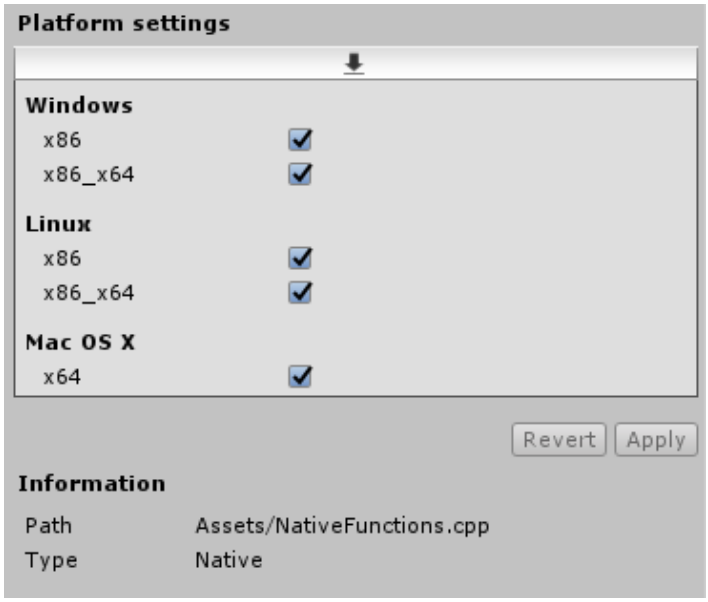
• 2018–03–13 Page published with editorial review

New feature in 2018.1

Windows Player: C++ source code plugins for IL2CPP

Leave feedback

You can add C++ (.cpp) code files directly into a Unity Project when using the IL2CPP **scripting backend**. These C++ files will act as plugins within the **Plugin Inspector**. If you configure the C++ files to be compatible with Windows Player, Unity compiles them together with C++ code that gets generated from managed assemblies. Click on your .cpp file and select the appropriate **Windows** options in the **Platform settings** section of the **Inspector** window:



Plugin importer settings for C++ files

Because the functions are linked together with generated C++ code, there is no separate DLL to `_P/Invoke` into. Due to this, you can use the "`__Internal`" keyword in place of the DLL name, which makes it the C++ linker's responsibility to resolve the functions, rather than loading them at run time, as the following example shows:

```
[DllImport("__Internal")]
private static extern int
CountLettersInString([MarshalAs(UnmanagedType.LPWSTR)]string str);
```

You can define this kind of function in NativeFunctions.cpp as follows:

```
extern "C" __declspec(dllexport) int __stdcall CountLettersInString(wchar_t* str)
{
    int length = 0;
    while (*str++ != nullptr)
        length++;
    return length;
}
```

Because the linker resolves the function call, any error made in the function declaration on the managed side (i.e. C# code that executes under managed run time) produces a linker error rather than a run-time error. This also means that no dynamic loading needs to take place during run time, and the function is called directly from C#. This significantly decreases the performance overhead of a `P/Invoke` call.

See Also

Plugin Inspector

• 2018–03–13 Page published with editorial review

New feature in 2018.1

Universal Windows Platform

Leave feedback

“Windows Store” platform renamed as “Universal Windows Platform” in Unity 2017.1

Getting Started

Leave feedback

Unity supports three CPU architectures when targeting **Universal Windows Platform**: X86, X64 and ARM.

There are three configurations (select them in Visual Studio):

Debug for debugging purposes
Release for profiling
Master for submission to store
The player log is located under <user>\AppData\Local\Packages\<productname>\TempState. See also log files.

Requirements when targeting Universal Windows Platform:

Windows 8.1
Visual Studio 2015 (RTM or later)
Windows 10 Universal SDK
Testing devices:
For basic testing of Windows SDK 10, same PC which is used for development is enough.
For basic testing of Phone 10, it's advisable to acquire a real Windows Phone 10 device.
Before you can proceed you need to acquire Windows Store developer license. There are two ways you can do this:

Build an empty Universal Windows application from Visual Studio and deploy it, if you're doing this first time, a dialog window for getting developer license should open, follow the steps.
See this link - http://msdn.microsoft.com/en-us/library/windows/apps/hh974578.aspx#getting_a_developer_license_by_using_visual_studio
The following are not supported:

Legacy Network classes (please use current Unity Networking). WWW and UnityWebRequest are supported (applies to all scripting backends).
`GameObject.SendMessage` partially works, but function which accepts the message must match the message sent, because the argument conversion doesn't work (applies only to .NET scripting backend).
You cannot automatically access C# classes from JS scripts (applies only to .NET scripting backend). To do this, go to PlayerSettings (menu: **File** > **Project Settings** > **Player**), open the **Publishing Settings**, navigate to the **Compilation** section and set the **Compilation Overrides** to **.NET Core Partially**.

Helpful links

Generic Universal Windows Apps samples https://github.com/Microsoft/Windows-universal-samples
Generic forum for Universal Windows Apps
https://social.msdn.microsoft.com/Forums/windowsapps/en-us/home?forum=wpdevelop
Unity forum dedicated to Windows development http://forum.unity3d.com/forums/50-Windows-Development

---

• 2017–05–16 Page amended with no editorial review

Universal Windows Platform: Deployment

Leave feedback

How to deploy Universal Windows Application

Go to Build Settings and build your project to a folder

Visual Studio solution will be generated

Running on **local machine**:
Make sure, that selected solution platform is x86 or x64 and **Debug target** is set to **“Local machine”**
Press F5 or go to **Debug** -> **Start debugging**
Running on **remote machine**:
If you are deploying to ARM architecture device, make sure that ARM is selected as **Solution Platform**
Start **Visual Studio Remote debugger** on a remote machine
Go to Visual Studio **project settings** -> **Debug**
Set target device to **“Remote machine”**
Find a remote machine in the list and select “Universal” authentication method
Press **F5** or go to **Debug** -> **Start debugging**
Note: If you cannot connect to a remote machine, make sure that Firewall does not block it
Running in **simulator**:
Set **Debug Target** to **“Simulator”**
Press F5 or go to **Debug** -> **Start debugging**

• 2017–05–16 Page amended with no editorial review

Universal Windows Platform: Profiler

Leave feedback

You can connect Unity's profiler to running Universal Windows Application. Perform the following steps:

Go to **Player Settings** > **Publishing Settings** > **Capabilities**
Enable Private Networks (Client & Server)
Build Universal Windows App Visual Studio solution from Unity.
Build and run the application
If you've checked **Autoconnect Profiler** checkbox, the profiler should connect automatically to Universal Windows App, if not, you have to explicitly pick it in Unity (**Window** > **Analysis** > **Profiler** > **Active Profiler**), for ex., MetroPlayerX86 (MyDevice)
Note: Profiler doesn't work on *Master* configuration

Note: Due **Universal Windows Platform** restrictions, you won't be able to connect the profiler if the application is running on the same machine. For ex., if you're running Unity editor and **Universal Windows Platform** on the same PC, you won't able to connect. You have to run Unity editor on one machine, and **Universal Windows Platform** on another machine. The only exception to this rule is "Autoconnect Profiler" build option, which makes the application connect to the editor instead.

Note: Also ensure that machine where Unity Editor is running and machine where Universal Windows App is running - **are on the same subnet**.

• 2017–05–16 Page amended with no editorial review

Universal Windows Platform: Command line arguments

Leave feedback

Universal Windows Apps don't accept command line arguments by default, so to pass them you have to call a special function from App.xaml.cs/cpp or App.cs/cpp. For ex., **appCallbacks.AddCommandLineArg("-nolog");** You have to call it before **appCallbacks.Initialize()** function.

Arguments

-nolog - Don't produce UnityPlayer.log.

-force-driver-type-warp - Force DirectX 11.0 WARP device (More info http://msdn.microsoft.com/en-us/library/gg615082.aspx)

-force-gfx-direct - Force single threaded **rendering**.

-force-d3d11-no-singlethreaded - Force DirectX 11.0 to be created without D3D11_CREATE_DEVICE_SINGLETHREADED flag.

-dontConnectAcceleratorEvent - Disable connecting to `AcceleratorKeyEvent`. This may help if you have issues with input in XAML elements. The downside is that Unity cannot handle some keys (for example, the keyboard keys **F10**, **Ctrl**, **Alt**, **Tab** may have issues in Unity).

-forceTextBoxBasedKeyboard - Use TextBox-based implementation for `TouchScreenKeyboard`. Only has an effect on UWP XAML applications. Allows switching to different implementations, in case there are issues with the default one.

DirectX feature levels

For further information about DirectX feature levels, see http://msdn.microsoft.com/en-us/library/windows/desktop/ff476876(v=vs.85).aspx)

-force-feature-level–9–1 - Force DirectX 11.0 feature level 9.1.

-force-feature-level–9–3 - Force DirectX 11.0 feature level 9.3.

-force-feature-level–10–0 - Force DirectX 11.0 feature level 10.0.

-force-feature-level–10–1 - Force DirectX 11.0 feature level 10.1.

-force-feature-level–11–0 - Force DirectX 11.0 feature level 11.0.

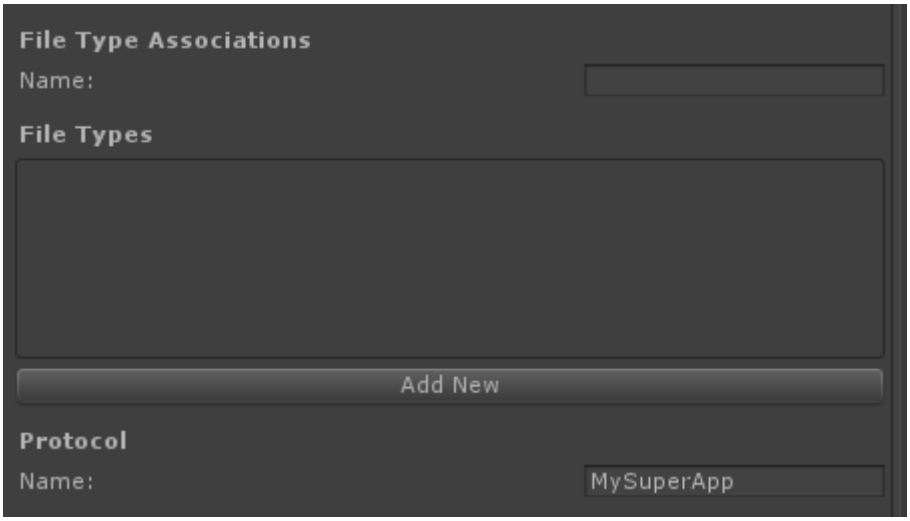
• 2017–05–16 Page amended with no editorial review

Universal Windows Platform: Association launching

Leave feedback

The API allows you to launch the user's default app for a file type or protocol. Check out - UnityEngine.WSA.Launcher for more information.

You can also enable your app to be the default app for a file type or protocol. You can set this information in Player Settings (More settings are available in **Package.appxmanifest** -> **Declarations**)



You can download an example project here.

You can read more here - Assocation launching.

• 2017–05–16 Page amended with no editorial review

AppCallbacks class

Leave feedback

You could call it a bridge between your main application and Unity engine. Here, we'll try to explain what every call to AppCallbacks exactly does. Let's build solution and explore App.xaml.cs file, which gets created if you use .NET **Scripting Backend**.

```
sealed partial class App : Application
{
    private WinRTBridge.WinRTBridge _bridge;
    private AppCallbacks appCallbacks;
    public App()
    {
        this.InitializeComponent();
        appCallbacks = new AppCallbacks(false);
    }

    protected override void OnLaunched(LaunchActivatedEventArgs args)
    {
        Frame rootFrame = Window.Current.Content as Frame;
        if (rootFrame == null)
        {
            var mainPage = new MainPage();
            Window.Current.Content = mainPage;
            Window.Current.Activate();

            _bridge = new WinRTBridge.WinRTBridge();
            appCallbacks.SetBridge(_bridge);

            appCallbacks.SetSwapChainBackgroundPanel(mainPage.GetSwapChainBackgroundPanel());

            appCallbacks.SetCoreWindowEvents(Window.Current.CoreWindow);

            appCallbacks.InitializeD3DXAML();
        }

        Window.Current.Activate();
    }
}
```

private WinRTBridge.WinRTBridge _bridge;

So first of all, what is WinRTBridge, it's used internally by Unity, to perform some of native-to-managed, managed-to-native operations, it's not intended to be used by developers. Due some WinRT platform restrictions, it cannot be created from Unity engine code, that's why WinRTBridge is being created here and is passed to Unity engine via appCallbacks.SetBridge(_bridge) instead.

appCallbacks = new AppCallbacks(false);

Now, let's take a closer look at AppCallbacks class. When you create it, you specify that your game will run on different thread, for backward compatibility reasons you can also specify that your application can run on **UI** thread, but that's not recommended, because there's a restriction from Microsoft - if your application won't become responsive after 5 seconds you'll fail to pass WACK (Windows Application Certification), read more here - http://msdn.microsoft.com/en-us/library/windowsphone/develop/hh184840(v=vs.105).aspx, imagine if your first level is pretty big, it might take significant amount of

time to load it, because your application is running on UI thread, UI will be unresponsive until your level is fully loaded. That's why it's recommend to always run your game on different thread.

Read more on UI thread here - http://msdn.microsoft.com/en-us/library/windows/apps/hh994635.aspx

Note: Code located in App.xaml.cs, MainPage.xaml.cs is always running on UI thread, unless called from InvokeOnAppThread function.

appCallbacks.SetSwapChainBackgroundPanel(mainPage.GetSwapChainBackgroundPanel());

This simply passes a XAML control to Unity which will be used as a render target for DirectX 11.

appCallbacks.SetCoreWindowEvents(Window.Current.CoreWindow);

Sets the core window for Unity, Unity subscribes to the following events (there may be more, depending on when this information was updated) :

VisibilityChanged
Closed
PointerCursor
SizeChanged
Activated
CharacterReceived
PointerPressed
PointerReleased
PointerMoved
PointerCaptureLost
PointerWheelChanged
AcceleratorKeyActivated

appCallbacks.InitializeD3DXAML();

This is main initialization function for Unity, it does following things:

Parse command line arguments, set by AppCallbacks.AddCommandLineArg()
Initialize DirectX 11 device
Load first level

At this point, when Unity finishes loading first level, it enters main loop.

Other functions

void InvokeOnAppThread(AppCallbackItem item, bool waitUntilDone)

Invokes a delegate on application thread, useful when you want to call your script function from UI thread.

void InvokeOnUIThread(AppCallbackItem item, bool waitUntilDone)

Invokes a delegate on UI thread, useful when you want to invoke something XAML specific API from your **scripts**.

bool RunningOnAppThread()

Returns true, if you're currently running in application thread.

bool RunningOnUIThread()

Returns true, if you're currently running in UI thread.

void InitializeD3DWindow()

Initialization function for D3D application.

void Run()

Function used by D3D application, for entering main loop.

bool IsInitialized()

Returns true, when first level is fully loaded.

void AddCommandLineArg(string arg)
Sets a command line argument for application, must be called before InitializeD3DWindow, InitializeD3DXAML.

void SetAppArguments(string arg) / string GetAppArguments()
Sets application arguments, which can be later accessed from Unity API - **UnityEngine.WSA.Application.arguments**.

void LoadGfxNativePlugin(string pluginFileName)
This function is obsolete and does nothing. In previous versions of Unity, this was needed to register native **plugins** for callbacks such as UnityRenderEvent. All **plugins** are now registered automatically. This function will be removed in a future update.

void ParseCommandLineArgsFromFiles(string fileName)
Parses command line arguments from a file, arguments must be separated by white spaces.

bool UnityPause(int pause)
Pauses Unity if you pass 1, unpauses if you pass 0, useful if you want to temporary freeze your game, for ex., when your game is snapped.

void UnitySetInput(bool enabled)
Enables/Disables input.

bool UnityGetInput()
Returns true, if Unity will process incoming input.

void SetKeyboardTriggerControl(Windows.UI.Xaml.Controls.Control ctrl)
Sets the control to be used for triggering on screen keyboard. This control will simply receive focus, when on screen keyboard is requested in scripts. Should be called with control, that does open keyboard on focus.

Windows.UI.Xaml.Controls.Control GetKeyboardTriggerControl()
Returns control, currently used control for triggering keyboard input. See SetKeyboardTriggerControl.

void SetCursor(Windows.UI.Core.CoreCursor cursor)
Sets system cursor. The given curosr is set for both CoreWindow and independent input source (if used).

void SetCustomCursor(unsigned int id)
Sets system cursor to custom. Parameter is cursor resource ID. Cursor is set for both CoreWindow and independent input source (if used).

Universal Windows Platform: WinRT API in C# scripts

Leave feedback

It is possible to use WinRT API directly in Unity **scripts**. However there are limitations and requirements for this:

Scripts must be written in C#
Scripts must be compiled using Microsoft's compiler, not Mono. This requires to set compilation overrides to **Use .NET Core** or **Use .NET Core Partially**, in the later case scripts must **not** be under **Plugins** or **Standard Assets** folders
Because the same script code is also used by Unity Editor (which always uses Mono), all code that uses WinRT API must be under **ENABLE_WINMD_SUPPORT** define
Below is an example for getting advertising using WinRT API directly:

```
using UnityEngine;
public class WinRTAPI : MonoBehaviour
{
    void Update()
    {
        auto adId = GetAdvertisingId();
        // ...
    }

    string GetAdvertisingId()
    {
        #if ENABLE_WINMD_SUPPORT
            return Windows.System.UserProfile.AdvertisingManager.AdvertisingId;
        #else
            return "";
        #endif
    }
}
```

Note: when using **IL2CPP scripting backend**, this is only supported when using .NET 4.6 compatiblity profile.

Universal Windows Platform Player Settings

Leave feedback

This page details the **Player Settings** specific to **Universal Windows Platform**. For a description of the general Player Settings, see documentation on Player Settings.

Most of these settings are transferred to the *Package.appxmanifest* file when creating Visual Studio solution for the first time.

Note: If you build your project on top of the existing one, Unity won't overwrite the *Package.appxmanifest* file if it's already present. That means if you change something in Player Settings, you should be sure to check *Package.appxmanifest*. If you want to regenerate *Package.appxmanifest*, delete it and rebuild your project from Unity.

To learn more, see Microsoft's documentation on App package manifest.

Settings from Packaging, Application **UI**, Tile, Splash screen, and Capabilities directly transfer to settings in the *Package.appxmanifest* file.

Supported orientations from Player Settings are also populated to the manifest (Package.appxmanifest file in Visual Studio solution). On Universal Windows Apps Unity will reset orientation to the one you used in Player Settings, regardless what you specify in the manifest. This is because Windows itself ignores those settings on desktop and tablet computers. Note, that you can always change supported orientations using Unity scripting API.

Certificate

Every Universal Windows App needs a certificate which identifies a developer, Unity will create a default certificate, if you won't provide your own.

Compilation

Unity uses Mono when compiling script files, and you can use the API located in .NET 4.x. **Compilation Overrides** allows you to use .NET for Universal Windows Platform (also known as .NET Core) in your C# files, the API is available here.

When Compilation Overrides is set to:

None - C# files are compiled using Mono compiler.
Use .Net Core - C# files are compiled using Microsoft compiler and .NET Core, you can use Windows Runtime API, but classes implemented in C# files aren't accessible from the JS language. Note: when using API from Windows Runtime, it's advisable to wrap the code with ENABLE_WINMD_SUPPORT define, because the API is only avaible when building to Universal Windows Platform, and it's not available in Unity Editor.
Use .Net Core Partially - C# files not located in **Plugins**, **Standard Assets**, Pro **Standard Assets** folders are compiled using Microsoft compiler and .NET Core, all other C# files are compiled using Mono compiler. The advantage is that classes implemented in C# are accessible from the JS language. Note: You won't be able to test .NET Core API in Unity Editor, because it doesn't have access to .NET Core, so you'll be able to test the API only when running Universal Windows App.
Note: You cannot use .NET Core API in JS **scripts**.

Here's a simple example of how to use .NET Core API in scripts.

```
string GetTemporaryFolder()
{
#if ENABLE_WINMD_SUPPORT
    return Windows.Storage.ApplicationData.Current.TemporaryFolder.Path;
#else
```

```
    return "LocalFolder";
#endif
}
```

Misc

Unprocessed Plugins contains a list of plugins which are ignored by Unity's preprocessing tools (like SerializationWeaver, AssemblyPreprocessor, rrw). Usually you don't need to modify this list, unless you're getting an error that Unity's failed to preprocess your plugin.

If you add a plugin to this list, Unity won't inject additional IL code into your assembly used for serialization purposes, but if your plugin isn't referencing UnityEngine.dll, that's fine, because Unity won't serialize any data from your plugin.

Independent Input Source

This allows you to enable an option for independent input source. Basically this makes your input more responsive, and usually you want this option to be enabled.

Low Latency Presentation API Lets you enable Low Latency Presentation API, basically this create D3D11 swapchain with DXGI_SWAP_CHAIN_FLAG_FRAME_LATENCY_WAITABLE_OBJECT flag, read more here and should increase input responsiveness. This option is disabled by default because on hardware with older GPU drivers, this option makes game laggy, if you enable this option - be sure to profile your game if the performance is still acceptable.

Capabilities

These options are directly copied to Package.appxmanifest.

Note: If you build your game on top of previous package, Package.appxmanifest won't be overwritten.

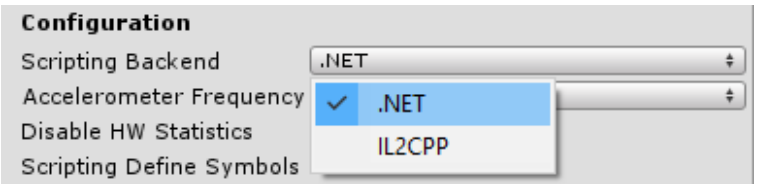
• 2017–05–16 Page amended with no editorial review

Scripting Backends

Leave feedback

Scripting backend is a framework that powers scripting in Unity. As of today, Unity supports three different **scripting backends** depending on target platform: Mono, .NET and **IL2CPP**. **Universal Windows Platform**, however, supports only two: .NET and **IL2CPP**.

You can switch between selected scripting backend in **player settings**:



• 2017–05–16 Page amended with no editorial review

Universal Windows Platform: .NET Scripting Backend

Leave feedback

The .NET **scripting backend** uses Microsoft's .NET to power scripting. **Universal Windows Platform** uses a subset of .NET 4.5. This is due to Windows Store certification requirements. This means that the platform has some incompatibilities compared to other platforms, such as a different **plugin** model and many missing types.

Note: The .NET scripting backend is deprecated and will be removed in a future release of Unity. We recommend switching to the IL2CPP scripting backend.

2018–08–03Page amended with limited editorial review

.NET scripting backend deprecated in 2018.2

.NET 3.5 scripting runtime deprecated in 2018.3

Universal Windows Platform: Generated project with .NET scripting backend

Leave feedback

Unity generates an XAML C# Visual Studio solution. With XAML C# solution generated you are able to use managed assemblies, such as UnityEngine.dll, Assembly-CSharp.dll, **etc**. with XAML code on top.

Unity will create files like resources, vcproj, xaml files. If you build a project on top of the same directory, these files will **not** be overwritten:

Project files & solution files (.vcproj, .sln, etc.)
Source files (App.xaml.cs, App.xaml.cpp)
XAML files (App.xaml, MainPage.xaml, etc.)
Image files (Assets\SmallTile.png, Assets\StoreLogo.png, etc.)
Manifest file - Package.appxmanifest
It is safe for you to modify these files, and if you want to revert to previous state, just remove the file, and build your project on top of the folder.

Note: Unity doesn't modify solution and project files if they already exist on the disk. Visual Studio takes the whole Data folder, instead of individual files, so if new file is added to Data folder, it's automatically picked.

Configurations

Generated Visual Studio projects have three configurations: **Debug**, **Release** and **Master**. **Debug** configuration has various safety checks and runs slower, **Release** configuration removes all those checks, but leaves **profiler** enabled. **Master** configuration should be used for final build that you submit to Store.

If you see **Development Build** text on the lower left corner, it means the game was not build for submission. This text will be shown, if you select **Development Build** when building form Unity, or if you've build **Debug** or **Release** configuration.

• 2017–05–16 Page amended with no editorial review

Universal Windows Platform: Missing .NET Types on .NET Scripting Backend

Leave feedback

The .NET **scripting backend** that is used on Universal Windows Apps is a special .NET version for this platform, which is not entirely compatible with Mono. In particular some data types are missing and some other classes don't have certain methods, that the same classes do have in Mono.

To make porting existing games to **Universal Windows Platform** easier, some of the missing .NET types are provided by Unity. In addition some extension methods and replacement types were added to make migration easier. These types are placed in **PlaybackEngines\metrosupport\Managed\WinRTLegacy.dll**.

Types, provided by Unity include:

System.Collections.ArrayList
System.Collections.Hashtable
System.Collections.Queue
System.Collections.SortedList
System.Collections.Stack
System.Collections.Specialized.HybridDictionary
System.Collections.Specialized.ListDictionary
System.Collections.Specialized.NameValueCollection
System.Collections.Specialized.OrderedDictionary
System.Collections.Specialized.StringCollection
System.IO.Directory
System.IO.File
System.IO.FileStream
System.Xml.XmlDocument
System.Xml.XmlTextReader
System.Xml.XmlTextWriter

In addition to these a namespace WinRTLegacy is added to provide additional classes and extention methods. Among there are:

Extention methods Close() for most System.IO classes (alternatively you can use Dispose(), which is available on both Mono and .NET for Universal Windows Platform)
WinRTLegacy.TypeExtensions has methods GetConstructor(), GetMethod(), GetProperty() for System.Type
WinRTLegacy.IO.StreamReader class, that is compatible with Mono System.IO.StreamReader
WinRTLegacy.IO.StreamWriter class, that is compatible with Mono System.IO.StreamWriter
WinRTLegacy.Xml.XmlReader class, that is compatible with Mono System.Xml.XmlReader
WinRTLegacy.Xml.XmlWriter class, that is compatible with Mono System.Xml.XmlWriter

The simplest way to use the replacement classes from WinRTLegacy if the namespace doesn't match is via using directive:

```
#if NETFX_CORE
using XmlReader = WinRTLegacy.Xml.XmlReader;
#else
```

```
using XmlReader = System.Xml.XmlReader;
#endif
```

This way you can use XmlReader class, which will be taken from WinRTLegacy.Xml namespace on Universal Windows Platform and from System.Xml namespace elsewhere.

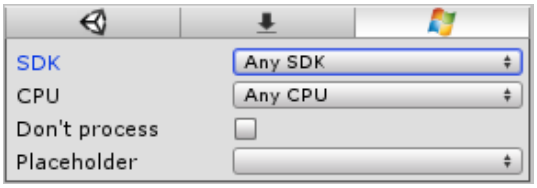
• 2017–05–16 Page amended with no editorial review

Universal Windows Platform: Plugins on .NET Scripting Backend

Leave feedback

Universal Windows Platform plug-in settings

To view these settings, go to the Unity Editor's Project Window, select the plug-in file, then in the Inspector window navigate to **Platform settings** > **Universal Windows Platform** (the Windows icon).



Universal Windows Platform plug-in settings

Property:	*Function:*
SDK	Use the drop-down to make the plug-in compatible with **Any SDK** or specific SDKs.
CPU	Use the drop-down to make the plug-in compatible with **Any CPU**, or limit the plug-in to **32-bit**, **64-bit** or **ARM** players.
Don't process (Only applies for managed assemblies)	Tick this checkbox to disable patching for this assembly. Assemblies need patching when they contain classes serializable by Unity. In these cases, Unity injects additional IL code into the assemblies. If you know the assemblies doesn't have these classes, then it's safe to disable patching. **Note:** Unity injects serialization code into your assemblies, so if you have a class derived from MonoBehaviour in your plug-in, and Unity doesn't patch it, you might get a serialization error during runtime.
Placeholder (Only applies for managed assemblies)	With Universal Windows Platform you can have plug-ins compiled against .NET Core, but because the Unity Editor runs on Mono, it fails to recognize these assemblies. As a result, C# files can't reference them. To work around this, you need to provide an assembly compiled against .NET 4.x with identical API, which acts as a placeholder for the real plugin (see next section, *Placeholder plug-ins*).

See documentation on the Plugin Inspector for more information.

Placeholder plug-ins

You cannot use Universal Windows Platform-specific **plugins** in the Unity Editor if you use Windows Runtime APIs. This section describes how the to handle this in the Unity Editor.

If you only intend to use the plugin for Universal Windows Platform, and not in the Unity Editor, you don't need to make a placeholder, but you do need to wrap the code which uses the plugin API with the following:

```
#if !UNITY_EDITOR
// Plugin code
#endif
```

If you intend to use the plugin for both Universal Windows Platform and the Unity Editor, you need a placeholder. Make two plugins:

For **Universal Windows Platform**, an assembly plug-in compiled against .NET Core with Windows Runtime API inside.
For the **Unity Editor**, an assembly plug-in compiled against .NET 4.5, which has identical public API with dummy function implementations (this is the placeholder).
Both plug-ins must share the same name and have the same assembly version. Note that the placeholder plugin for the Unity Editor cannot reference *UnityEditor.dll*. if it does, Unity generates an error.

The steps below describe how to assign a platform to each in the Editor.

In the Unity Editor's Project window, select your Editor-compatible placeholder plug-in. In the **Inspector** window, go to **Select platforms for plugin** and select **Editor** as the only compatible platform.

In the Unity Editor's Project window, select your Universal Windows Platform-compatible placeholder plug-in. In the Inspector window, go to **Select platforms for plugin** and select **Universal Windows Platform** as the only compatible platform.

In the Inspector window for the Universal Windows Platform-compatible plug-in, set the **Placeholder** field to your Editor-compatible placeholder plug-in.

This means that when building to Universal Windows Platform, Unity uses Editor-compatible placeholder plug-in when compiling your **scripts**, but copies the Universal Windows Platform-compatible plug-in to the final folder. This achieves two things: The Unity Editor successfully compiles your **scripts**, but the built game itself still uses the API from the Universal Windows Platform-specific plug-in.

• 2018–08–03 Page amended with limited editorial review

.NET 3.5 scripting runtime deprecated in 2018.3

Universal Windows Platform: Debugging on .NET Scripting Backend

Leave feedback

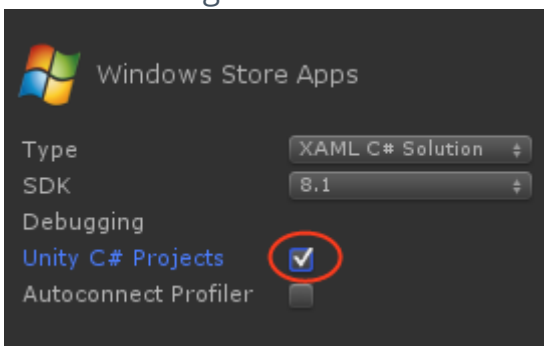
When you have a crash, or a weird behavior, always check the player log which is located here - **<user>\AppData\Local\Packages<productname>\TempState\UnityPlayer.log**. When submitting a bug, please include the player log as well, it can give invaluable information.

Currently it's only possible to debug C# **scripts**.

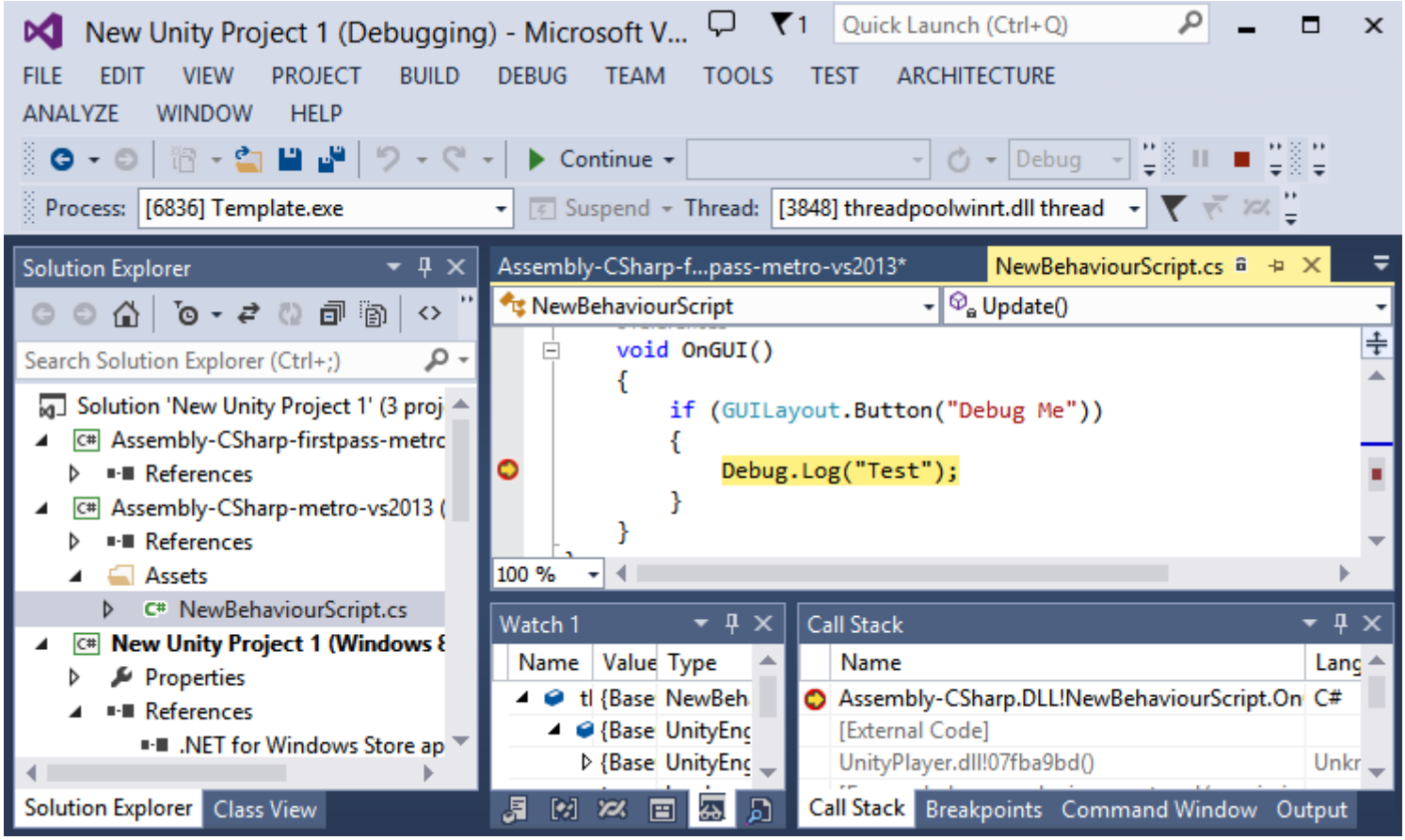
Note: **Universal Windows Platform** are running with Microsoft .NET, that's why it's not possible to debug scripts with **MonoDevelop**, instead you have to use Visual Studio 2013.

Here are simple steps how to do it:

When building to Universal Windows Platform, check **Unity C# projects**



This will create Assembly-CSharp-* projects compatible with Universal Windows Apps
Note: if previously **Unity C# projects** was unchecked, build to empty folder or delete *.sln and* .csproj, because Unity needs to add references to those files, but if they'll be present - Unity won't overwrite them
Open the solution, you should see Assembly-CSharp-* projects included
Place breakpoints in places of interest, and simply start application with the debugger



Exceptions

When you run the application, you can tell Visual Studio to stop during exception. Go to **Debug** -> **Exceptions**:

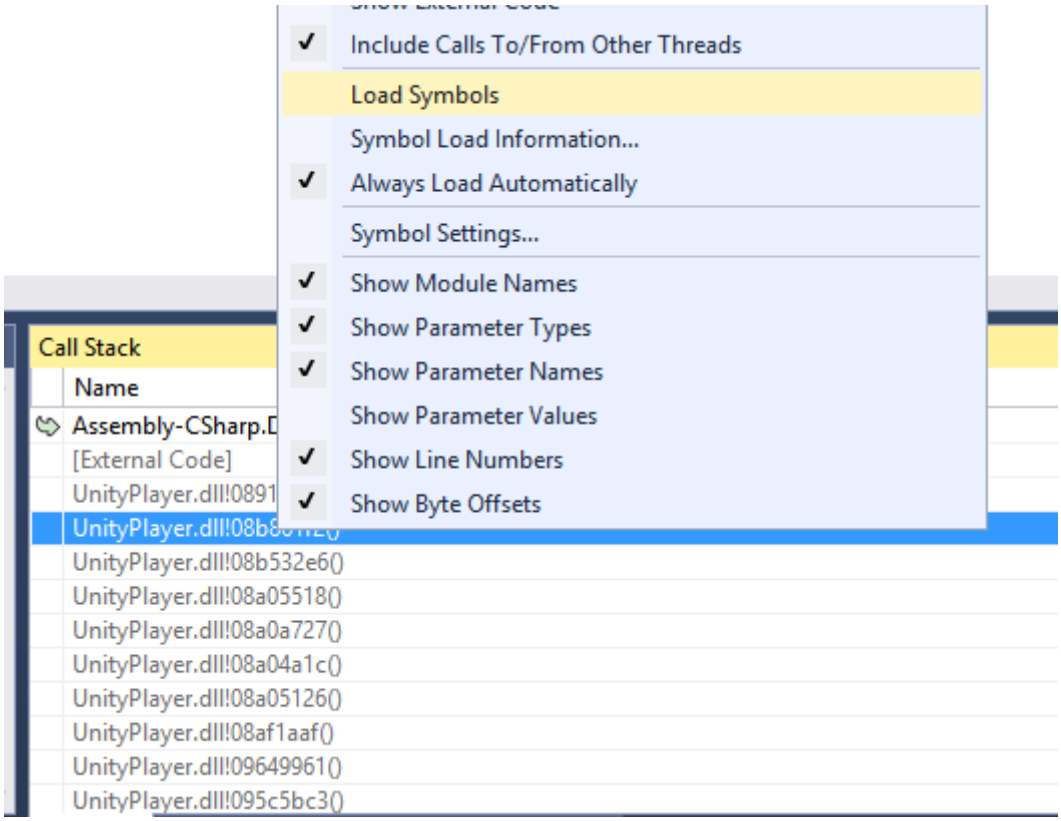
Enable Common Language Runtime Exceptions and Managed Debugging Assistants - for managed exceptions
Enable all exceptions if you're getting exception in some unknown place
Note: enabling all exceptions will make Visual Studio to stop even at the harmless exceptions like **WinRT originate error**, **WinRT transform error**, ignore those and simply continue

Resolving callstack from UnityPlayer.dll

There will be cases when you'll have a crash in Unity engine itself, you can get useful information if you're able to resolve the callstack, and provide it in the bug report if needed.

Note: Callstacks from Unity engine can be resolved if *.pdb files are available, Unity provides* .pdb files for Debug configuration.

Suppose you've encountered a crash in Unity engine and hit the breakpoint (**Note**: Visual Studio can stop at the crash if you enable all exceptions via **Debug** -> **Exceptions** menu), go to **Debug** -> **Windows** -> **Call Stack**, Call Stack window should open up, if you don't see function names from UnityPlayer.dll, that means the symbols weren't loaded, to fix that, right click on that function and Load Symbols, UnityPlayer.pdb file will be located in [PathToYourProject]>\Players\[Windows80 or Windows81]\[X86 or ARM or X64]\debug.



Microsoft-Windows-TWinUI

This is a log which might provide information why your application didn't launch without debugger, it can be found in:

Control Panel -> **Administrative Tools** -> **Event Viewer** -> **Applications And Services Log** -> **Microsoft** -> **Windows** -> **Apps** -> **Microsoft-Windows-TWinUI/Operational**

• 2017–05–16 Page amended with no editorial review

Universal Windows Platform: IL2CPP scripting back end

Leave feedback

IL2CPP is a fully-supported scripting back end that can be used as an alternative to .NET when building projects for **Universal Windows Platform**. When building a project using IL2CPP, Unity converts IL code from **scripts** and assemblies to C++ before creating a native binary. Refer to the Unity Manual page on IL2CPP for more information.

• 2017–05–16 Page amended with no editorial review

Universal Windows Platform: Generated project with IL2CPP scripting backend

Leave feedback

Building a project from Unity to **Universal Windows Platform** with **IL2CPP scripting backend** will create a Visual Studio C++ solution containing three projects:

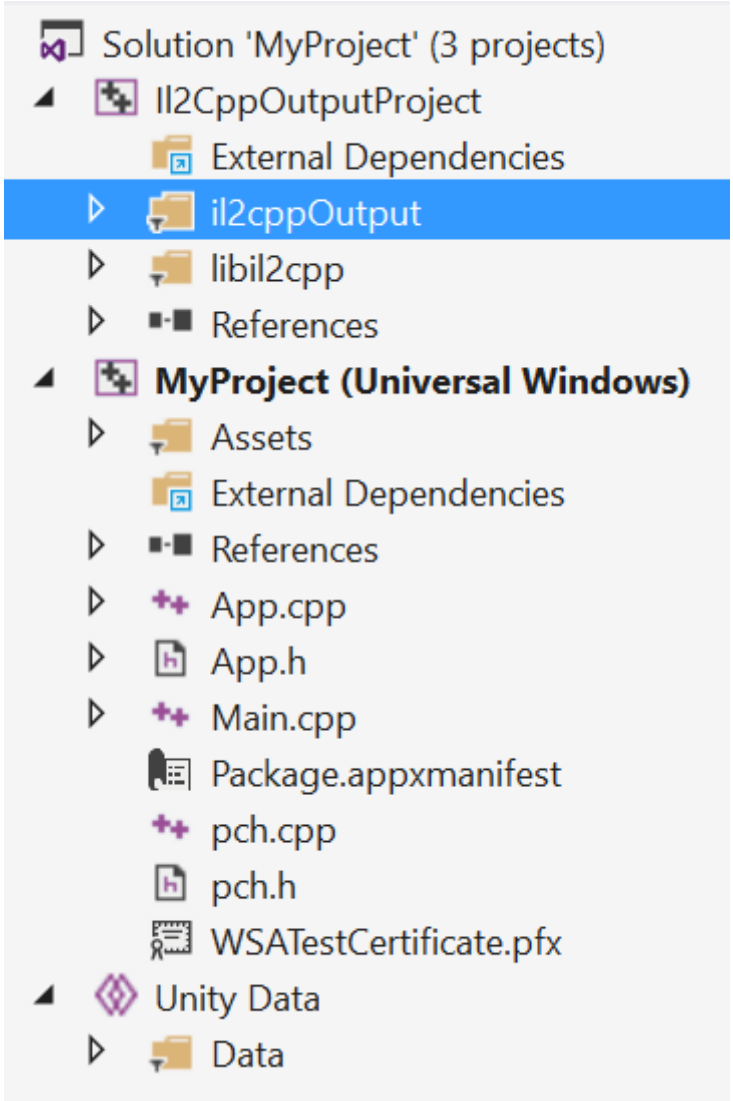


Figure 1. Generated Visual Studio solution

These projects serve their own purposes:

Il2CppOutputProject contains generated C++ code that was converted from managed assemblies. This project gets overwritten every time when built over it. See figure 2.
Unity Data project contains all Unity data files: levels, **assets**, **etc**. This project gets overwrriten every time as well.
The main project (it name matches your Unity project name). This is the project that will be built into an application package, which may be deployed to a device or uploaded to the Windows Store. Unity will not overwrite this project when built on top of it, so it can be modified freely without the fear of changes becoming lost.

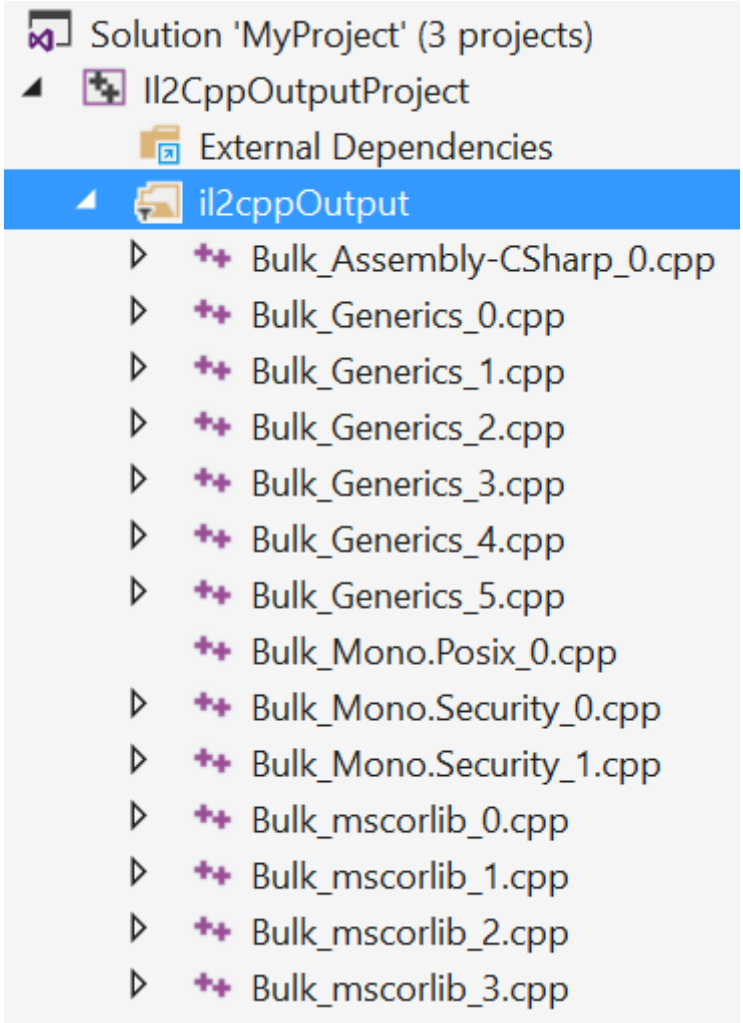


Figure 2. Il2CppOutputProject

When using .NET scripting backend, the generated project is using C#, however, this scenario is unsupported when using IL2CPP scripting backend.

Configurations

Generated Visual Studio projects have three configurations: **Debug**, **Release** and **Master**:

Debug configuration has all optimizations disabled, all debugging info preserved and runs much slower. It is used for debugging your game.
Release configuration enables most code optimizations, but leaves **profiler** enabled. This configuration is used for profiling your game.
Master configuration disables the profiler, and is used for game submission/final testing. The build time of **Master** configuration might be longer, however, it is a little bit faster than Release configuration.

• 2017–05–16 Page amended with no editorial review

Universal Windows Platform: Plugins on IL2CPP Scripting Backend

Leave feedback

At this point in time, the **plugin** model for **Universal Windows Platform** with **IL2CPP scripting backend** is much more similar to other Unity platforms (such as Windows standalone), rather than Universal Windows Platform with .NET scripting backend.

Managed plugins

By default, IL2CPP targets .NET 2.0 API compatiblity level. That means unlike .NET scripting backend, it does not support **managed plugins** targeting .NET 4.5 or consuming any of Windows Runtime APIs. All **managed plugins** must be targeting .NET 4.5 or equivalent API when using this compatiblity level. You can switch to .NET 4.6 API compatibilty level in **player settings** if you wish to lift these restrictions.

Note The .Net 3.5 scripting runtime is deprecated, but it will be available for Unity 2018.3 and 2018.4 LTS. Please migrate or start new Projects with the .NET 4.x runtime.

Another difference compared to .NET scripting backend is that IL2CPP scripting backend exposes the exact same .NET API surface as Unity editor or standalone player, so it's possible to use the same plugins without the need to compile separate versions target different .NET API for Universal Windows Platform.

Native plugins

IL2CPP scripting backend supports using **native plugins** through P/Invoke mechanism. It means that you can call into **native plugins** directly from your C# code by specifying the native function prototype and then calling it. For example:

```
[DllImport("MyPlugin.dll")]
private static extern int CountLettersInString([MarshalAs(UnmanagedType.LPWSTR)]string

private void Start()
{
    Debug.Log(CountLettersInString("Hello, native plugin!"));
}
```

The implementation of such function inside MyPlugin.dll would look like this:

```
extern "C" __declspec(dllexport)
int __stdcall CountLettersInString(wchar_t* str)
{
    int length = 0;
    while (*str++ != nullptr)
        length++;
    return length;
}
```

P/Invoke marshaling rules match that of official .NET marshaling, with exception of few unsupported types:

AnsiBStr
BStr
Currency
SAFEARRAY
IDispatch
IUnknown
TBStr
VBByRefStr

The default calling convention for P/Invoke functions on x86 is `__stdcall`.

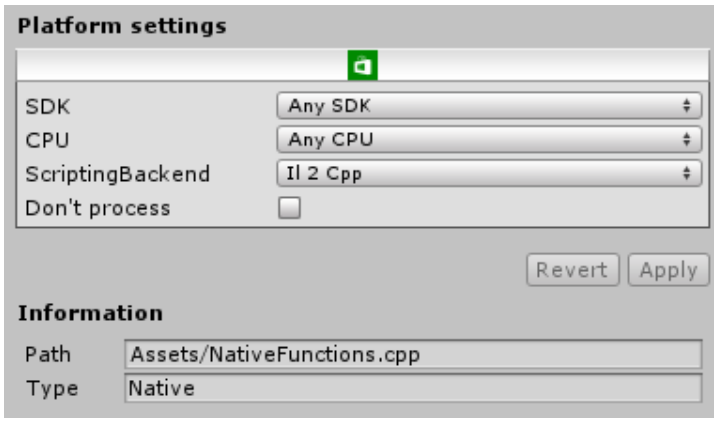
Native plugins can be authored in two ways: precompiled DLL or C++ source code.

Precompiled native plugins

P/Invoking into precompiled native plugins works by loading the DLL at runtime, finding the function entry point and then calling it. These DLLs must be compiled against appropriate Windows SDK for the target CPU architecture. The DLLs must also be configured in Plugin **Inspector** when added to Unity project.

C++ source code native plugins

It is possible to add C++ (.cpp) code files directly into Unity project, which will act as a plugin in Plugin Inspector. If configured to be compatible with Universal Windows Platform and IL2CPP scriping backend, these C++ files will be compiled together with C++ code that gets generated from managed assemblies:



Since the functions are linked together with generated C++ code, there is no separate DLL to P/Invoke into. Due to this, it's possible to use "__Internal" keyword in place of DLL name, which will make it C++ Linker responsibility to resolve the functions, rather than loading them at runtime:

```
[DllImport("__Internal")]
private static extern int CountLettersInString([MarshalAs(UnmanagedType.LPWSTR)]string
```

Since the call is resolved by the linker, making an error in function declaration on managed side will produce a linker error, rather an error at runtime. This also means that no dynamic loading needs to take place at runtime, and function is called directly. This significantly decreases the overhead of a P/Invoke call.

P/Invoke limitations

On Universal Windows Platform you cannot P/Invoke into specific system libraries by specifying the dll name (like “kernelbase.dll”) when using IL2CPP scripting backend. Attempting to P/Invoke into any DLL that exists outside of the project will result in DllNotFoundException at runtime.

However, it’s still possible to P/Invoke into these system functions by specifying “**Internal” keyword instead of the DLL name, which results in linker resolving the functions at build time.

See also

Plugin Inspector

• 2018–08–03 Page amended with no editorial review

.NET 3.5 scripting runtime deprecated in 2018.3

Universal Windows Platform: Debugging on IL2CPP Scripting Backend

Leave feedback

Although **IL2CPP** does not have a C# debugger at this time, it's still possible to debug generated C++ code using Visual Studio.

Class and method naming in generated C++ code

IL2CPP classes look like this: `<ClassName>_t#number`, where `<ClassName>` is plain name of a class, while #number is a unique type number. #number is not present on some of the core types. For example:

```
String_t
Object_t
Type_t
Char_t34
StringBuilder_t26
GuidParser_t1539
```

IL2CPP methods look like this: `<ClassName>_<MethodName>_m#number`, where `<ClassName>` is the plain class name of methods declaring type, `<MethodName>` is a plain method name and `#number` is unique method number. For example:

```
GuidParser_ParseHex_m10003
ConfigurationSection_DoDeserializeSection_m1275
String_Format_m4102
Mathf_Sqrt_m289
Thing_Start_m1
```

Static field structures are named like this: `<ClassName>_t#number_StaticFields`, where the first part of the structure name is identical to the declaring type, for example:

```
StringBuilder_t26_StaticFields
Thing_t24_StaticFields
```

Furthermore, above each class/method definition there is a C++ comment stating full class/method name. For example:

```
// System.String
struct String_t : public Object_t
{
    // System.Int32 System.String::length
    int32_t _length_0;
    // System.Char System.String::start_char
    uint16_t _start_char_1;
};

// System.Text.StringBuilder
struct StringBuilder_t26  : public Object_t
{
    // System.Int32 System.Text.StringBuilder::_length
    int32_t length_1;
    // System.String System.Text.StringBuilder::_str
    String_t* str_2;
    // System.String System.Text.StringBuilder::_cached_str
    String_t* cached_str_3;
    // System.Int32 System.Text.StringBuilder::_maxCapacity
    int32_t maxCapacity_4;
};

// System.Void MyData::.ctor()
extern "C" void MyData_ctor_m0 (MyData_t2 * this, const MethodInfo* method)
{
    ...
}

// Thing
struct Thing_t24 : public MonoBehaviour_t25
{
    // MyData Thing::m_Data
    MyData_t2 * _m_Data_2;
    // System.Text.StringBuilder Thing::m_Builder
    StringBuilder_t26 * _m_Builder_3;
};
struct Thing_t24_StaticFields
{
    // System.Int32 Thing::s_SomeStaticField
    int32_t _s_SomeStaticField_4;
};
```

Observing variable values

One of the most important parts of debugging is observing values of various variables. Visual Studio allows to do that relatively easily by either mousing over the variable of adding it into the watch window. For example:

```
350         StringBuilder_AppendFormat_m12(L_8, _stringLiteral2, L_12, /*hidden argument*/NULL);
351         StringBuilder_t26 * L_13 = (__this->___m_Builder_3);
352         MyData_t2 * L_14 = ___data;
353         NullCheck(L_14);
354         uint8_t L_15 = (L_14->___ByteValue_3);
355         uint8_t L_16 = L_15;
356         Object_t * L_16 21 '\x15' 36_il2cpp_TypeInfo_var, &L_16);
357         NullCheck(L_13);
358         StringBuilder_AppendFormat_m12(L_13, _stringLiteral3, L_17, /*hidden argument*/NULL);
359         StringBuilder_t26 * L_18 = (__this->___m_Builder_3);
360         MyData_t2 * L_19 = ___data;
```

Watch 1

Name	Value	Type
L_15	21 '\x15'	unsigned char
__this	0x0000004bdabaafc0 {__m_Data_2=0x0000004bdabade40 {__Strir	Thing_t24 *
MonoBehaviour_t25	{...}	MonoBehaviour_t25
__m_Data_2	0x0000004bdabade40 {__StringValue_0=0x0000004bdaba3f00 L"la	MyData_t2 *
__m_Builder_3	0x0000004bdabaaf80 {___length_1=4141 ___str_2=0x0000004be15	StringBuilder_t26 *
L_14	0x0000004bdabadd10 {__StringValue_0=0x0000004bdaba3e80 L"la	MyData_t2 *
Object_t	{_typeInfo=0x0000004be5990a00 {image=0x0000004be2b0dec0 {na	Object_t
__StringValue_0	0x0000004bdaba3e80 L"lauke karšta"	String_t *
__CharValue_1	279	unsigned short
__IntValue_2	-84	int
__ByteValue_3	21 '\x15'	unsigned char
__FloatValue_4	3.40000010	float
__LongValue_5	841	__int64
__StructValue_6	{__ShortValue_0=-45 __DoubleValue_1=849849.11111000006 }	SimpleStruct_t1
__MatrixValue_7	{__m00_0=7.00000000 __m10_1=5.00000000 __m20_2=3.000000	Matrix4x4_t3

Observing static fields is a little bit harder. In IL2CPP, static fields are stored on a Il2CppClass instance itself. So in order to observe a static field, we'll first need a pointer to Il2CppClass structure of that type. These pointers are in scope of methods that use them, but after observing it once, it will remain at the same memory address for the duration of application run. Il2CppClass structure has "static_fields" field, which is a pointer to a memory block containing static fields for that particular type. To view the actual values, this pointer has to be cast to appropriate static field structure: each type has its own. For example, let's observe the static fields of class `Thing_t24`:

Watch 1

Name	Value	Type
(Thing_t24_StaticFields*)Thing_t24_il2cpp_TypeInfo_var->static_fields	0x000000fd4f821d60 {__s_SomeStaticField_4=	Thing_t24_StaticFields *
__s_SomeStaticField_4	3	int

Investigating exceptions

IL2CPP uses native C++ exceptions to implement .NET exceptions. When any exception is supposed to be thrown, IL2CPP throws Il2CppExceptionWrapper object, which is defined as the following:

```
struct Il2CppExceptionWrapper
{
    Il2CppException* ex;

    Il2CppExceptionWrapper (Il2CppException* ex) : ex (ex) {}
};
```

These exception objects can be easily investigated in the watch window:

Watch 1

Name	Value	Type
e	{ex=0x000000c7ddf0f850 {object={_typeInfo=0x000000c7e78f7470 {image=0x00000	Il2CppExceptionWrapper &l
ex	0x000000c7ddf0f850 {object={_typeInfo=0x000000c7e78f7470 {image=0x000000c7e	Il2CppException *
object	{_typeInfo=0x000000c7e78f7470 {image=0x000000c7e8e0e12c {nameIndex=138993	Il2CppObject
trace_ips	0x000000c7e1928500 {data={dummy=0x0000000000000454 klassIndex=1108 type=0	Il2CppArray *
inner_ex	0x0000000000000000 <NULL>	Il2CppObject *
message	0x000000c7e021ce80 L"Hello there"	Il2CppString *
help_link	0x0000000000000000 <NULL>	Il2CppString *
class_name	0x0000000000000000 <NULL>	Il2CppString *
stack_trace	0x0000000000000000 <NULL>	Il2CppString *
remote_stack_trace	0x0000000000000000 <NULL>	Il2CppString *
remote_stack_index	0	int
hresult	-2147024809	int
source	0x0000000000000000 <NULL>	Il2CppString *
_data	0x0000000000000000 <NULL>	Il2CppObject *

Lastly, it might be beneficial to enable debugger-break on exception so the source of the exception can be caught red-handed. To do so, hit CTRL+ALT+E in Visual Studio, and make sure that C++ exceptions checkbox is checked in the opened window:

Exception Settings

Search

Break When Thrown

- [x] C++ Exceptions
- [ ] Common Language Runtime Exceptions
- [ ] GPU Memory Access Exceptions
- [ ] JavaScript Runtime Exceptions
- [ ] Managed Debugging Assistants
- [ ] WebKit JavaScript Exceptions
- [ ] Win32 Exceptions

Threads | Memory 1 | Modules | Call Stack | Breakpoints | Exception Settings | Command Window | Immediate Window | Output

After enabling this setting, Visual Studio will automatically stop execution whenever an exception is thrown:

```
21  void Exception::Raise (Il2CppException* ex)
22  {
23      if (ex->trace_ips == NULL)
24      {
25          // Only write the stack trace if there is not one already in the exception.
26          // When we exit managed try/finally and try/catch blocks with an exception, this m
27          // called with the original exception which already has the proper stack trace.
28          // Getting the stack trace again here will lose the frames between the original th
29          // and the finally or catch block.
30          const StackFrames& frames = *StackTrace::GetStackFrames();
31          size_t i = frames.size() - 1;
32          Il2CppArray* ips = Array::New(il2cpp_defaults.int_class, (il2cpp_array_size_t)fram
33          for (StackFrames::const_iterator iter = frames.begin(); iter != frames.end(); ++it
34          {
35              il2cpp_array_set(ips, const MethodInfo*, i, (*iter).method);
36          }
37
38          IL2CPP_OBJECT_SETREF(ex, trace_ips, ips);
39      }
40
41      throw Il2CppExceptionWrapper (ex);
42  }
43
44  void Exception::RaiseOutOfMemoryException ()
45  {
46      Raise (GetOutOfMemoryException ());
```

Microsoft Visual Studio

Exception thrown at 0x00007FFBB494A1C8 in JSONSerializeTest.exe: Microsoft C++ exception: Il2CppExceptionWrapper at memory location 0x000000687F5FE9F0.

If there is a handler for this exception, the program may be safely continued.

☑ Break when this exception type is thrown

Break and open Exception Settings

Break | Continue | Ignore

Value	Type
identifier "e" is undefined	

Call Stack

Name	Lang
UnityPlayer.dll!il2cpp::vm::Exception::Raise(Il2CppException * ex) Line 41	C++
UnityPlayer.dll!il2cpp_codegen_raise_exception(Exception_t27 * ex) Line 144	C++
UnityPlayer.dll!Thing_Start_m1(Thing_t24 * __this, const MethodInfo * method) Line 185	C++
UnityPlayer.dll!RuntimeInvoker_Void_t854(const MethodInfo * method, void * obj, void * * args) Line 396	C++
UnityPlayer.dll!il2cpp::vm::Runtime::Invoke(const MethodInfo * method, void * obj, void * * params, Il2CppObject * * exc) Line 328	C++
UnityPlayer.dll!il2cpp_runtime_invoke(const MethodInfo * method, void * obj, void * * params, Il2CppObject * * exc) Line 870	C++
UnityPlayer.dll!il2cpp_invoke_method(ScriptingMethodIl2Cpp method, ScriptingObject * object, ScriptingArguments * arguments, ScriptingExce	C++
UnityPlayer.dll!ScriptingInvocation::Invoke(ScriptingException * * exception, bool convertArguments) Line 176	C++
UnityPlayer.dll!ScriptingInvocation::Invoke(ScriptingException * * exception) Line 50	C++
UnityPlayer.dll!MonoBehaviour::InvokeMethodOrCoroutineChecked(ScriptingMethodIl2Cpp scriptingMethod, ScriptingObject * value, Scripting	C++
UnityPlayer.dll!MonoBehaviour::InvokeMethodOrCoroutineChecked(ScriptingMethodIl2Cpp method, ScriptingObject * value) Line 859	C++
UnityPlayer.dll!MonoBehaviour::Start() Line 486	C++
UnityPlayer.dll!MonoBehaviour::DelayedStartCall(Object * o, void * userData) Line 1231	C++
UnityPlayer.dll!DelayedCallManager::Update(int modeMask) Line 180	C++
UnityPlayer.dll!PlayerLoop(bool batchMode, bool performRendering, IHookEvent * pHookEvt) Line 1563	C++

Threads | Memory 1 | Modules | Call Stack | Breakpoints | Exception Settings | Command Window | Immediate Window | Output

---

• 2017–05–16 Page amended with no editorial review

FAQ

Leave feedback

Unsupported classes and functions when using .NET Scripting Backend

There are many classes and methods that are not supported in .NET for Universal Windows Apps, see this link for supported classes: http://msdn.microsoft.com/en-us/library/windows/apps/br230232.aspx.

How to create AppX package from Visual Studio ?

After building the project from Unity Editor open it with Visual Studio
In the solution explorer, right click on the project
Store -> **Create App Packages**
Do you want to build packages to upload to the Windows Store? Choose No, then Next
Pick appropriate platform, for ex., ARM Release
Don't include public symbol files, this will make package smaller
Create
Locate folder which is named something like YourApp_1.0.0.0_ARM_Test, check that it has **Add-AppDevPackage.ps1** file
Copy the folder contents to the target machine, then on the target machine right click on **Add-AppDevPackage.ps1** -> **Run with PowerShell**
Follow the steps, you might need an internet connection to install Developper License, this will require for you to have Microsoft account
If everything is okay, your app should appear on the start menu

How to install an appx file on your machine?

Open **Windows PowerShell** from start menu, navigate to your appx file, execute **Add-AppxPackage <yourappx>.appx**, if the appx was signed, it will be installed on your machine. **Note:** if you're installing appx file again, you have to uninstall the previous one, simply right-click on the icon, and click Uninstall.

I am getting an error "DEP0600: incorrect parameter" while deploying an application.

Something is wrong with your certificate, try creating a new by clicking on **Package.appxmanifest** -> **Packaging** -> **Choose Certificate** -> **Configure Certificate** -> **Create Test Certificate**

How to use Visual Studio's graphical debugger on ARM?

Read this post http://msdn.microsoft.com/en-us/library/hh780905(v=vs.110).aspx . You will need to create a WinRT **component**, and reference it from main module.

How to deploy a project on a tablet PC?

See Deployment.

How do I choose which compiler to use for my C# scripts?

Under publishing settings on **Universal Windows Platform player settings**, there's a drop down menu called "Compilation overrides". There are 3 settings:

```
1. None. All C# __scripts__ will get compiled with Mono C# compiler;
2. Use Net Core Partially. Scripts that are in folders "__Assets__/__Plugins__",
```

```
3. Use Net Core. All scripts will get compiled with Microsoft C# compiler.
```

Both compilers have their ups and downs. Compiling scripts with the Mono C# compiler will allow them to be referenced by JavaScript scripts, which, for example, is needed for Angry Bots (hence you have to set it to none). However, using the Microsoft C# compiler will allow you to use Microsoft specific APIs without the need for plugins - just wrap the code in #if ENABLE_WINMD_SUPPORT/#endif, and it will compile and work just fine.

Getting more information about Windows App Certification Kit (WACK) failure?

You can find a log in <user>\AppData\Local\Microsoft\AppCertKit which might contain additional information about the failure.

Help! There's too many defines! Which are defined when?

No worries. Here's all of them:

UNITY_WINRT	Defined on all scripts
UNITY_WSA	Defined on all scripts
UNITY_WINRT_10_0	Defined on all scripts
UNITY_WSA_10_0	Defined on all scripts
ENABLE_DOTNET	Defined on all scripts when using .NET **scripting backend**
ENABLE_IL2CPP	Defined on all scripts when using **IL2CPP** scripting backend
NETFX_CORE	Defined on C# scripts that are compiled using Microsoft C# compiler when using .NET scripting backend
WINDOWS_UWP	Defined on C# scripts that are compiled using Microsoft C# compiler when using .NET scripting backend or IL2CPP scripting backend with .NET 4.6 compatibility level
ENABLE_WINMD_SUPPORT	Defined on C# scripts that are compiled using Microsoft C# compiler when using .NET scripting backend or IL2CPP scripting backend with .NET 4.6 compatibility level

See also platform dependent compilation.

Breakpoints in generated Assembly-CSharp-* projects aren't hit.

There could be couple of reasons:

This may occur because of the JIT optimization on module load. In Visual Studio, go to **Tools** > **Options** > **Debugging** > **General** and uncheck **Suppress JIT optimization on module load**. Visual Studio doesn't consider Assembly-CSharp-* as your code. Go to **Tools** > **Options** > **Debugging** > **General** and uncheck **Enable Just My Code**. This tells Visual Studio that you want to debug the Assembly-CSharp-* projects.

• 2017–05–16 Page amended with no editorial review

Universal Windows Platform: Examples

Leave feedback

Example projects are stored on the Unity Technologies Bitbucket.

• 2017–05–16 Page amended with editorial review

Universal Windows Platform: Code snippets

Leave feedback

How to to implement “Rate Us” functionality?

Associate these lines with button click.

```
#if ENABLE_WINMD_SUPPORT
            string productID =  Windows.ApplicationModel.Package.Current.Id.Fami
            Application.OpenURL("ms-windows-store://pdp/?ProductId=" + productID
#endif
```

• 2017–05–16 Page amended with no editorial review

Known issues

Leave feedback

This page contains known issues with apps built for the **Universal Windows Platform** caused by external factors that aren't related to Unity, such as drivers and libraries.

Setup	Issue	Reason	Workaround
Nokia Lumia 630/635/1520	Specific sampler ordering in the **Shader** causes the Texture Wrap mode to change to Clamp mode. This might also be reproduced on other 3xx Adreno devices.	Adreno driver bug.	Change the sampler register for the affected Texture in the **Shader** code file (for example, change `sampler2D _MainTex;` to `sampler2D _MainTex:register(s0);`).
UWP	Scrolling the mouse wheel can break mouse movement if the cursor is locked (if Cursor.lockState `!= None`).	Windows OS issue.	Open the **Player Settings** (**Edit** > **Project Settings** > **Player**). Select the **Publishing Settings** tab, and tick the **Independent Input Source** checkbox.
UWP .Net Native	The functions `Terrain.GetHeights`, `Terrain.SetHeights`, and `Terrain.SetHeightsDelayLOD` crash the player.	.Net Native bug when calling `GCHandle.Alloc` with multi-dimensional arrays.	Currently no workaround.

• 2017–05–16 Page amended with no editorial review

Web Player

Leave feedback

The Web Player feature was removed from 5.4. It is no longer supported. If you need Web Player support please use a version of Unity prior to 5.4. There is no way to use Unity for Web Player content beyond 5.3.

Mobile Developer Checklist

Leave feedback

If you are having problems when developing for a mobile platform, this is a checklist to help you solve various problems.

Crashes

Leave feedback

Checklist for crashes

Disable code stripping (and set “slow with exceptions” for iOS)
Follow the instructions on Optimizing the Size of the Built **iOS** Player (iphone-playerSizeOptimization) to make sure your game does not crash with stripping on iOS.
Verify it is not because of out of memory (restart your device, use the device with maximum RAM for the platform, be sure to watch the logs)

Editor.log - on the editor

The Debug messages, warnings and errors all go to the console. Also Unity prints status reports to the console - loading **assets**, initializing mono, graphics driver info.

If you are trying to understand what is going on look at the editor.log. Here you will get the full picture, not just a console fragment. You can try to understand what’s happening, and watch the full log of your coding session. This will help you track down what has caused Unity crash to crash or find out what’s wrong with your assets.

Unity prints some things on the devices as well; Logcat console for Android and Xcode gdb console on iOS devices

Debugging on Android

Use the *DDMS* or *ADB* tool
Watch the stacktrace (Android 3 or newer). Either use *c++filt* (part of the *ndk*) or the other methods, like: http://slush.warosu.org/c++filtjs to decode the mangled function calls
Look at the *.so* file that the crash occurs on:
libunity.so - the crash is in the Unity code or the user code
libdvm.so - the crash is in the Java world, somewhere with Dalvik. So find Dalvik’s stacktrace, look at your JNI code or anything Java-related (including your possible changes to the *AndroidManifest.xml*).
libmono.so - either a Mono bug or you’re doing something Mono strongly dislikes
If the crashlog does not help you can disassemble it to get a rough understanding of what has happened.
use ARM EABI tools from the Android NDK like this: *objdump.exe -S libmono.so >> out.txt*
Look at the code around pc from the stacktrace.
try to match that code within the fresh *out.txt* file.
Scroll up to understand what is happening in the function it occurs in.

Debugging on iOS

Xcode has built in tools. Xcode 4 has a really nice GUI for debugging crashes, Xcode 3 has less.

Full gdb stack - thread backtrace all

Enable soft-null-check: Enable **development build** and script debugging. Now uncaught null ref exceptions will be printed to the Xcode console with the appropriate managed call stack.

Try turning the “fast script call” and code stripping off. It may stop some random crashes, like those caused by using some rare *.Net* functions or reflection.

Strategy

Try to figure out which script the crash happens in and debug it using mono develop on the device. If the crash seems to not be in your code, take a closer look at the stacktrace, there should be a hint of something happening. Take a copy and submit it, and we'll take a look.

Profiling

Leave feedback

First steps

Unity relies on the CPU (heavily optimized for the SIMD part of it, like SSE on x86 or NEON on ARM) for **skinning**, batching, physics, user **scripts**, particles, **etc**.

The GPU is used for **shaders**, drawcalls, image effects.

CPU or GPU bound

Use the internal **profiler** to detect the CPU and GPU ms

Pareto analysis

A large majority of problems (80%) are produced by a few key causes (20%). You can use the Editor profiler to identify the most processor-intensive function calls and optimize them first. Often, optimizing a few key functions can give a significant increase in overall execution speed.

You should make sure that functions are only executed when really necessary. For example, you can use events like *OnBecameVisible* and *OnBecameInvisible* to detect when an object can't be seen and avoid updating it. Coroutines can be a useful way to call code that needs regular updates but doesn't need to run every frame:-

```
// Do some stuff every frame:
void Update () {
}

//Do some stuff every 0.2 seconds:
IEnumerator SlowUpdate () {
   while (true) {
      //do something
      yield return new WaitForSeconds (0.2f);
   }
}
```

You can use the .NET *System.Threading.Thread* class to run heavy calculations on a separate thread. This allows you to run on multiple cores but note that the Unity API is not thread-safe - you need to buffer inputs and results and read and assign them on the main thread in order to use Unity API calls.

CPU Profiling

Profile user code

Not all of the user code is shown in the Profiler. But you can use *Profiler.BeginSample* and *Profiler.EndSample* to make the required user code appear in the profiler.

GPU Profiling

The Unity Editor profiler cannot show GPU data as of now. We're working with hardware manufacturers to make it happen with the Tegra devices being the first to appear in the Editor profiler.

Tools for iOS

Unity internal profiler (not the Editor profiler). This shows the GPU time for the whole **scene**.
PowerVR PVRUniSCo shader analyzer. See below.
iOS: Xcode OpenGL ES Driver Instruments can show only high-level info:
"Device Utilization %" - GPU time spent on rendering in total. >95% means the app is GPU bound.
"Renderer Utilization %" - GPU time spent drawing **pixels**.
"Tiler Utilization %" - GPU time spent processing vertices.
"Split count" - the number of frame splits, where the vertex data didn't fit into allocated buffers.
PowerVR is tile based deferred renderer, so it's impossible to get GPU timings per draw call. However you can get GPU times for the whole scene using Unity's built-in profiler (the one that prints results to Xcode output). Apple's tools currently can only tell you how busy the GPU and its parts are, but do not give times in milliseconds.

PVRUniSCo gives cycles for the whole shader, and approximate cycles for each line in the shader code. Windows & Mac! But it won't match what Apple's drivers are doing exactly anyway. Still, a good ballpark measure.

Tools for Android

Adreno (Qualcomm)
NVPerfHUD (NVIDIA)
PVRTune, PVRUniSCo (PowerVR)
On Tegra, NVIDIA provides excellent performance tools which does everything you want - GPU time per draw call, Cycles per shader, Force 2x2 texture, Null view rectangle, runs on Windows, OSX, Linux. PerfHUD ES does not easily work with consumer devices, you need the development board from NVIDIA.

Qualcomm provides excellent Adreno Profiler (Windows only) which is Windows only, but works with consumer devices! It features **Timeline** graphs, frame capture, Frame debug, API calls, Shader analyzer, live editing.

Graphics related CPU profiling

The internal profiler gives a good overview per module:

time spent in OpenGL ES API
batching efficiency
skinning, animations, particles

Profiler Ports

Ports that the Unity profiler uses:

MulticastPort : 54998
ListenPorts : 55000 - 55511
Multicast (unittests) : 55512 - 56023
They should be accessible from within the network node. That is, the devices that you're trying to profile on should be able to see these ports on the machine with the Unity Editor with the Profiler on.

Memory

There are two types of memory, Mono memory and Unity memory.

Mono memory

Mono memory handles script objects, wrappers for Unity objects (game objects, assets, components, etc). Garbage Collector cleans up when the allocation does not fit in the available memory or on a *System.GC.Collect()* call.

Memory is allocated in heap blocks. More can allocated if it cannot fit the data into the allocated block. Heap blocks will be kept in Mono until the app is closed. In other words, Mono does not release any memory used to the OS (Unity 3.x). Once you allocate a certain amount of memory, it is reserved for mono and not available for the OS. Even when you release it, it will become available internally for Mono only and not for the OS. The heap memory value in the Profiler will only increase, never decrease.

If the system cannot fit new data into the allocated heap block, the Mono calls a “GC” and can allocate a new heap block (for example, due to fragmentation).

“Too many heap sections” means you’ve run out of Mono memory (because of fragmentation or heavy usage).

Use `System.GC.GetTotalMemory` to get the total used Mono memory.

The general advice is, use as small an allocation as possible.

Unity memory

Unity memory handles **Asset** data (Textures, Meshes, Audio, Animation, etc), Game objects, Engine internals (Rendering, Particles, Physics, etc). Use `Profiler.usedHeapSize` to get the total used Unity memory.

Memory map

No tools yet but you can use the following.

Unity Profiler - not perfect, skips stuff, but you can get an overview. It works on the device!
Internal profiler. Shows Used heap and allocated heap - see mono memory. Shows the number of mono allocations per frame.
Xcode tools - iOS
Xcode Instruments Activity Monitor - Real Memory column.
Xcode Instruments Allocations - net allocations for created and living objects.
VM Tracker (textures usually get allocated with IOKit label and meshes usually go into VM Allocate).
You can also make your own tool using Unity API calls:-

```
FindObjectsOfTypeAll (type : Type) : Object[]
FindObjectsOfType (type : Type): Object[]
GetRuntimeMemorySize (o : Object) : int
GetMonoHeapSize
GetMonoUsedSize
Profiler.BeginSample/EndSample - profile your own code
UnloadUnusedAssets () : AsyncOperation
System.GC.GetTotalMemory/Profiler.usedHeapSize
```

References to the loaded objects - There is no way to figure this out. A workaround is to “Find references in scene” for public variables.

Garbage collector

This fires when the system cannot fit new data into the allocated heap block.
Don’t use `OnGUI()` on mobiles: it shoots several times per frame, completely redraws the view and creates tons of memory allocation calls that require Garbage Collection to be invoked.

Creating/removing too many objects too quickly?

This may lead to fragmentation.
Use the Editor profiler to track the memory activity.
The internal profiler can be used to track the mono memory activity.
`System.GC.Collect()` You can use this *.Net* function when it’s ok to have a hiccup.

Allocation hiccups

Use lists of preallocated, reusable class instances to implement your own memory management scheme.
Don’t make huge allocations per frame, cache, preallocate instead
Problems with fragmentation?

Preallocating a memory pool.

Keep a List of inactive **GameObjects** and reuse them instead of Instantiating and Destroying them.

Out of mono memory

Profile memory activity - when does the first memory page fill up?
Do you really need so many gameobjects that a single memory page is not enough?
Use structs instead of classes for local data. Classes are stored on the heap; structs on the stack.
Read the Understanding Automatic Memory Management page.

Out of memory crashes

At some points a game may crash with “out of memory” though it in theory it should fit in fine. When this happens compare your normal game memory footprint and the allocated memory size when the crash happens. If the numbers are not similar, then there is a memory spike. This might be due to:

Two big scenes being loaded at the same time - use an empty scene between two bigger ones to fix this.
Additive scene loading - remove unused parts to maintain the memory size.
Huge asset bundles loaded to the memory
Textures without proper **compression** (a no go for mobiles).
Textures having Get/Set pixels enabled. This requires an uncompressed copy of the texture in memory.
Textures loaded from JPEG/PNGs at runtime are essentially uncompressed.
Big mp3 files marked as decompress on loading.
Keeping unused assets in weird caches like static monobehavior fields, which are not cleared when changing scenes.

Optimizations

Leave feedback

Just like on PCs, mobile platforms like **iOS** and Android have devices of various levels of performance. You can easily find a phone that's 10x more powerful for **rendering** than some other phone. Quite easy way of scaling:

Make sure it runs okay on baseline configuration
Use more eye-candy on higher performing configurations:
Resolution
Post-processing
MSAA
Anisotropy
Shaders
Fx/particles density, on/off

Focus on GPUs

Graphics performance is bound by fillrate, **pixel** and geometric complexity (vertex count). All three of these can be reduced if you can find a way to cull more renderers. **Occlusion culling** could help here as Unity will automatically cull objects outside the viewing frustum.

On mobiles you're essentially fillrate bound (fillrate = screen pixels * shader complexity * overdraw), and over-complex **shaders** is the most common cause of problems. So use mobile **shaders** that come with Unity or design your own but make them as simple as possible. If possible simplify your pixel **shaders** by moving code to **vertex shader**.

If reducing the Texture Quality in Quality Settings makes the game run faster, you are probably limited by memory bandwidth. So compress textures, use mipmaps, reduce texture size, **etc**.

LOD (Level of Detail) - make objects simpler or eliminate them completely as they move further away.

Good practice

Mobile GPUs have huge constraints in how much heat they produce, how much power they use, and how large or noisy they can be. So compared to the desktop parts, mobile GPUs have way less bandwidth, low ALU performance and texturing power. The architectures of the GPUs are also tuned to use as little bandwidth & power as possible.

Unity is optimized for OpenGL ES 2.0, it uses GLSL ES (similar to HLSL) shading language. Built in shaders are most often written in HLSL (also known as Cg). This is cross compiled into GLSL ES for mobile platforms. You can also write GLSL directly if you want to, but doing that limits you to OpenGL-like platforms (e.g. mobile + Mac) since there currently are no GLSL->HLSL translation tools. When you use float/half/fixed types in HLSL, they end up highp/mediump/lowp precision qualifiers in GLSL ES.

Here is the checklist for good practice:

Keep the number of materials as low as possible. This makes it easier for Unity to batch stuff.
Use texture atlases (large images containing a collection of sub-images) instead of a number of individual textures. These are faster to load, have fewer state switches, and are batching friendly.
Use *Renderer.sharedMaterial* instead of *Renderer.material* if using texture atlases and shared materials.
Forward rendered pixel lights are expensive.
Use light mapping instead of realtime lights where ever possible.
Adjust pixel light count in quality settings. Essentially only the directional light should be per pixel, everything else - per vertex. Certainly this depends on the game.
Experiment with Render Mode of Lights in the Quality Settings to get the correct priority.
Avoid Cutout (alpha test) shaders unless really necessary.
Keep Transparent (alpha blend) screen coverage to a minimum.
Try to avoid situations where multiple lights illuminate any given object.
Try to reduce the overall number of shader passes (Shadows, pixel lights, reflections).
Rendering order is critical. In general case:

fully opaque objects roughly front-to-back.
alpha tested objects roughly front-to-back.
skybox.
alpha blended objects (back to front if needed).
Post Processing is expensive on mobiles, use with care.
Particles: reduce overdraw, use the simplest possible shaders.
Double buffer for Meshes modified every frame:

```
void Update (){
  // flip between meshes
  bufferMesh = on ? meshA : meshB;
  on = !on;
  bufferMesh.vertices = vertices; // modification to mesh
  meshFilter.sharedMesh = bufferMesh;
}
```

Shader optimizations

Checking if you are fillrate-bound is easy: does the game run faster if you decrease the display resolution? If yes, you are limited by fillrate.

Try reducing shader complexity by the following methods:

Avoid alpha-testing shaders; instead use alpha-blended versions.
Use simple, optimized shader code (such as the "Mobile" shaders that ship with Unity).
Avoid expensive math functions in shader code (pow, exp, log, cos, sin, tan, etc). Consider using pre-calculated lookup textures instead.
Pick lowest possible number precision format (float, half, fixedin Cg) for best performance.

Focus on CPUs

It is often the case that games are limited by the GPU on pixel processing. So they end up having unused CPU power, especially on multicore mobile CPUs. So it is often sensible to pull some work off the GPU and put it onto the CPU instead (Unity does all of these): **mesh skinning**, batching of small objects, particle geometry updates.

These should be used with care, not blindly. If you are not bound by draw calls, then batching is actually worse for performance, as it makes culling less efficient and makes more objects affected by lights!

Good practice

FindObjectsOfType (and Unity getter properties in general) are very slow, so use them sensibly.
Set the Static property on non-moving objects to allow internal optimizations like **static batching**.
Spend lots of CPU cycles to do occlusion culling and better sorting (to take advantage of Early Z-cull).

Physics

Physics can be CPU heavy. It can be profiled via the Editor **profiler**. If Physics appears to take too much time on CPU:

Tweak *Time.fixedDeltaTime* (in Project settings -> Time) to be as high as you can get away with. If your game is slow moving, you probably need less fixed updates than games with fast action. Fast paced games will need more frequent calculations, and thus *fixedDeltaTime* will need to be lower or a collision may fail.
Physics.solverIterationCount (Physics Manager).
Use as little Cloth objects as possible.
Use **Rigidbodies** only where necessary.

Use primitive **colliders** in preference **mesh colliders**.
Never ever move a static collider (ie a collider without a Rigidbody) as it causes a big performance hit. It shows up in Profiler as "Static Collider.Move' but actual processing is in *Physics.Simulate*. If necessary, add a RigidBody and set *isKinematic* to true.
On Windows you can use NVidia's AgPerfMon profiling tool set to get more details if needed.

Android

GPU

These are the popular mobile architectures. This is both different hardware vendors than in PC/console space, and very different GPU architectures than the "usual" GPUs.

ImgTec PowerVR SGX - Tile based, deferred: render everything in small tiles (as 16x16), shade only visible pixels
NVIDIA Tegra - Classic: Render everything
Qualcomm Adreno - Tiled: Render everything in tile, engineered in large tiles (as 256k). Adreno 3xx can switch to traditional.
ARM Mali Tiled: Render everything in tile, engineered in small tiles (as 16x16)
Spend some time looking into different rendering approaches and design your game accordingly. Pay especial attention to sorting. Define the lowest end supported devices early in the dev cycle. Test on them with the profiler on as you design your game.

Use platform specific texture **compression**.

Further reading

PowerVR SGX Architecture Guide http://imgtec.com/powervr/insider/powervr-sdk-docs.asp
Tegra GLES2 feature guide
http://developer.download.nvidia.com/assets/mobile/files/tegra_gles2_development.pdf
Qualcomm Adreno GLES performance guide
http://developer.qualcomm.com/file/607/adreno200performanceoptimizationopenglestipsandtricksmarch10.pdf
Engel, Rible http://altdevblogaday.com/2011/08/04/programming-the-xperia-play-gpu-by-wolfgang-engel-and-maurice-ribble/
ARM Mali GPU Optimization guide

Screen resolution

Android version

iOS

GPU

Only PowerVR architecture (tile based deferred) to be concerned about.

ImgTec PowerVR SGX. Tile based, deferred: render everything in tiles, shade only visible pixels.
This means:

Mipmaps are not so necessary.
Antialiasing and aniso are cheap enough, not needed on iPad 3 in some cases.
And cons:

If vertex data per frame (number of vertices * storage required after vertex shader) exceeds the internal buffers allocated by the driver, the **scene** has to be "split" which costs performance. The driver might allocate a larger buffer after this point, or you might need to reduce your vertex count. This becomes apparent on iPad2 (iOS 4.3) at around 100 thousand vertices with quite complex shaders.
TBDR needs more transistors allocated for the tiling and deferred parts, leaving conceptually less transistors for "raw performance". It's very hard (i.e. practically impossible) to get GPU timing for a draw call on TBDR, making

profiling hard.

Further reading

PowerVR SGX Architecture Guide http://imgtec.com/powervr/insider/powervr-sdk-docs.asp

Screen resolution

iOS version

Dynamic Objects

Asset Bundles

Asset Bundles are cached on a device to a certain limit
Create using the Editor API
Load using WWW API: WWW.LoadFromCacheOrDownload or as a resource: AssetBundle.CreateFromMemory or AssetBundle.CreateFromFile
Unload using AssetBundle.Unload. There is an option to unload the bundle, but keep the loaded asset from it. Also can kill all the loaded assets even if they're referenced in the scene
Resources.UnloadUnusedAssets unloads all assets no longer referenced in the scene. So remember to kill references to the assets you don't need. Public and static variables are never garbage collected.
Resources.UnloadAsset unloads a specific asset from memory. It can be reloaded from disk if needed.

Is there any limitation for download numbers of Assetbundle at the same time on iOS? (e.g Can we download over 10 assetbundles safely at the same time(or every frame)?)

Downloads are implemented via async API provided by OS, so OS decides how many threads need to be created for downloads. When launching multiple concurrent downloads you should keep in mind total device bandwidth it can support and amount of free memory. Each concurrent download allocates its own temporal buffer, so you should be careful there to not run out of memory.

Resources

Assets need to be recognized by Unity to be placed in a build.
Add .bytes file extension to any raw bytes you want Unity to recognize as a binary data.
Add .txt file extension to any text files you want Unity to recognize as a text asset
Resources are converted to a platform format at a build time.
Resources.Load()

Issue checklist

Textures without proper compression
Different solutions for different cases, but be sure to compress textures unless you're sure you should not.
ETC/RGBA16 - default for android but can tweak depending on the GPU vendor. Best approach is to use ETC where possible. Alpha textures can use two ETC files with one channel being for alpha
PVRTC - default for iOS, good for most cases
Textures having Get/Set pixels enabled - doubles the footprint, uncheck unless Get/Set is needed
Textures loaded from JPEG/PNGs on the runtime will be uncompressed
Big mp3 files marked as decompress on load
Additive scene loading
Unused Assets that remain uncleaned in memory.
If it randomly crashes, try on a devkit or a device with 2 GB memory (like Ipad 3).
Sometimes there's nothing in the console, just a random crash

Fast script call and stripping may lead to random crashes on iOS. Try without them.

Practical guide to optimization for mobiles

Leave feedback

This guide is for developers new to mobile game development, who are probably feeling overwhelmed and are either planning and prototyping a new mobile game or porting an existing project to run smoothly on a mobile device. This guide should also be useful as a reference for anyone making mobile games or browser games that target old PCs and netbooks.

Optimization is a broad topic, and how you do it depends a lot on your game. Because of this, this guide is best read as an introduction or reference rather than a step-by-step guide that guarantees a smooth product.

All mobile devices are not created equal

The information here assumes hardware around the level of the Apple A4 chipset, which is used on the original iPad, the iPhone 3GS, and the third generation iPod Touch. On the Android side, that would mean an Android phone such as the Nexus One, or most phones that run Android 2.3 Gingerbread. Most of these devices were released around early 2010. Out of the app-hungry market, these devices are the older, slower portion, but they should be supported because they represent a large portion of the market.

For an overview of Apple mobile device tech specs, see documentation on iPhone hardware. The very low-end Apple mobile devices (such as the iPhone 3G) and the first and second generation iPod Touches are extremely limited, and even more care must be taken to optimize for them. However, there is some question as to whether consumers who have not upgraded their device will be buying apps at all. So, unless you are making a free app, it might not be worthwhile to support the old hardware.

There are much slower and much faster phones out there, and the computational capability of mobile devices is increasing at an extraordinary rate. It's not unheard of for a new generation of a mobile GPU to be five times faster than its predecessor. That's incredibly fast when compared to the PC industry.

Make optimization a design consideration, not a final step

British computer scientist Michael A. Jackson is often quoted for his rules of program optimization:

"The first rule of program optimization: don't do it. The second rule of program optimization (for experts only!): don't do it yet."

His rationale was that, considering how fast computers are and how quickly their speed is increasing, there is a good chance that, if you program something, it will run fast enough. Additionally, if you try to optimize too heavily, you might over-complicate things, limit yourself, or create bugs.

However, if you are developing mobile games, there is another consideration: The hardware that is on the market right now is very limited compared to the computers we are used to working with, so the risk of creating something that simply won't run on the device balances out the risk of over-complication that comes with optimizing from the start.

Throughout this guide, we will try to point out situations where an optimization would help a lot, versus situations where it would just be frivolous.

Optimization: not just for programmers

Artists also need to know the limitations of the platform, and the methods that are used to get around them, so they can make creative choices that pay off without having to re-produce work.

More responsibility can fall on the artist if the game design calls for atmosphere and lighting to be drawn into Textures instead of being baked.
Whenever anything can be baked, artists can produce content for baking instead of real-time **rendering**. This allows them to ignore technical limitations and work freely.

Design your game for a smooth runtime

These two pages detail general trends in game performance, and explain how you can best design your game to be optimized or how you can intuitively figure out which things need to be optimized if you've already gone into production.

Practical Methods for Optimized Rendering
Practical Methods for Optimized Scripting and Gameplay

Profile early and often

Profiling is important because it helps you discern which optimizations will pay off with big performance increases and which ones are a waste of your time. Because of the way that rendering is handled on a separate chip (GPU), the time it takes to render a frame is not the time that the CPU takes plus the time that the GPU takes. Instead, it is the longer of the two.

That means that if the CPU is slowing things down, optimizing your **Shaders** won't increase the frame rate at all, and if the GPU is slowing things down, optimizing physics and **scripts** won't help at all.

Often, different parts of the game and different situations perform differently as well. This means one part of the game might cause 100 millisecond frames entirely due to a script, and another part of the game might cause the same slowdown but because of something that is being rendered. At the very least, you need to know where all the bottlenecks are if you're going to optimize your game.

Unity Profiler

You can use the main Profiler in Unity when targeting **iOS**, Android or Tizen. See documentation on the Profiler for basic instructions on how to use it.

Internal profilers

Andriod and iOS both have a built-in internal profiler, which spews out text every 30 frames. It can help you figure out which aspects of your game are slowing things down (such as physics, scripts, or rendering), but it doesn't go into much detail (for example, it can't tell you which script or renderer is the culprit).

If the profiler indicates that most of your processing time is spent in rendering, see documentation on Rendering Optimizations
If the profiler indicates that most of your processing time is spent outside of rendering, see documentation on Scripting Optimizations
See documentation on Internal profilers for information on how they work and how to turn them on.

Table of contents

Practical Guide to Optimization for Mobiles - Graphics Methods
Practical Guide to Optimization for Mobiles - Scripting and Gameplay Methods
Practical Guide to Optimization for Mobiles - Rendering Optimizations
Practical Guide to Optimization for Mobiles - Optimizing Scripts

Graphics Methods

Leave feedback

What are mobile devices capable of? How should you plan your game accordingly? If your game runs slow, and the **profiler** indicates that it's a **rendering** bottleneck, how do you know what to change, and how to make your game look good but still run fast? This page is dedicated to a general and non-technical exposition of the methods. If you are looking for the specifics, see the Rendering Optimizations page.

What you can reasonably expect to run on current consumer mobiles:

Lightmapped static geometry. But beware of:
Using a lot of alpha-test shaders
Bumpmapping, especially using built-in **shaders**.
High polygon count
Animated characters, even with fancy shaders! But beware of:
Massive crowds or high-poly characters
2D games with **sprites**. But beware of:
Overdraw, or, lots of layers drawn on top of each other.
Particle effects. But beware of:
High density on large particles. (Lots of particles drawn on top of each other. This is another overdraw situation)
Ridiculous numbers of particles, or particle **colliders**.
Physics. But beware of:
Mesh colliders.
Lots of active bodies.

What you CANNOT reasonably expect to run on current consumer mobiles:

Fullscreen screen image effects like glow and **depth of field**.
Dynamic per-pixel lighting (multiple lights marked Important and not baked into the lightmap)
Every affected object is drawn an additional time for every dynamic light you use, and this gets slow quickly.
Real time shadows on everything
Unity has support for real time shadows on mobile platforms, but their use must be very judicious, and likely limited to higher-end devices.

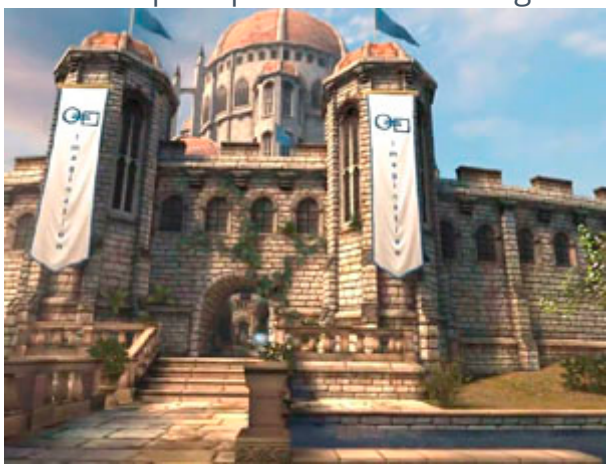
Examples - How top-notch mobile games are made

Shadowgun

Shadowgun is an impressive example of what can be done on current mobile hardware. But more specifically, it's a good example of what cannot be done, and how to get around the limitations. Especially because a small part of the game has been made publicly available in this blog post.

Here's a basic rundown of things that Shadowgun does in order to keep performance up:

Dynamic lighting - barely used.
Blob shadows and **Lightmaps** are used instead of any real shadows.
Lightprobes, instead of real lights, are used on their characters.
Muzzle flashes added into the lightprobe data via script.
The only dynamic per-pixel lighting is an arbitrary light direction used to calculate a BRDF on the characters.
Bumpmapping - barely used.
Real bumpmapping only used on characters.
As much contrast and detail as possible is baked into the diffuse texture maps. Lighting information from bumpmaps is baked in.
A good example is their statue texture, or their shiny wall, as seen on the right. No bumpmaps are used to render these, the specularity is faked by baking it into the texture. Lightmapping is combined with a vertex-lighting-based specular highlight to give these models a shiny look.
If you want to learn how to create textures like this one, check out the Rendering Optimizations page.
Dense particles - avoided.
UV-scrolling textures used instead of dense particle effects.
Fog effects - avoided.
Their god rays are hand-modeled.
Single planes that fade in and out are used to achieve cinematic fog effects without actually rendering any fog.
This is faster because the planes are few and far between, and it means that fog doesn't have to be calculated on every **pixel** and in every shader.
Glow - avoided.
Blended sprite planes are used to give the appearance of a glow on certain objects.



Sky Castle Demo

This demo was designed to show what Unity is capable of on high-end Android devices.

Dynamic lighting - not used.

Lightmaps only.
Bumpmapping - used
The bricks are all bumpmapped, lit by directional lightmaps. This is where the “high-end devices” part comes into play.
Real time reflections - limited.
They carefully placed their real-time reflecting surfaces separately and in isolated areas, so that only one runs at a time, and the environment that needs to be rendered twice can be easily culled.

Bottom line - What this means for your game

The more you respect and understand the limitations of the mobile devices, the better your game will look, and the smoother it will perform. If you want to make a high-class game for mobile, you will benefit from understanding Unity’s graphics pipeline and being able to write your own shaders. But if you want something to grab to use right away, ShadowGun’s shaders, available here, are a good place to start.

Don’t Simulate It, Bake It!

There is no question that games attempt to follow the laws of nature. The movement of every parabolic projectile and the color of every pixel of shiny chrome is derived by formulas first written to mimic observations of the real world. But a game is one part scientific simulation and one part painting. You can’t compete in the mobile market with physically accurate rendering; the hardware simply isn’t there yet, if you try to imitate the real world all the way, your game will end up limited, drab, and laggy.

You have to pick up your polygons and your blend modes like they’re paintbrushes.

The baked bumpmaps shown in Shadowgun are great examples of this. There are specular highlights already in the texture - the human eye doesn’t notice that they don’t actually line up with the reflected light and view directions - they are simply high-contrast details on the texture, completely faked, yet they end up looking great. This is a common cheating technique which has been used in many successful games. Compare the visor in the first Halo screenshot ever released with the visor from this release screenshot. It appears that the armor protrusions from the top of the helmet are reflected in the visor, but the reflection is actually baked into the visor texture. In League of Legends, a spell effect appears to have a pixel-light attached to it, but it actually is a blended plane with a texture that was probably generated by taking a screenshot of a pixel light shining on the ground.

What works well:

Lightmapped static geometry
Dramatic lighting and largely dynamic environments don’t mix. Pick one or the other.
Lightprobes for moving objects
Current mobile hardware is not really cut out for lots of dynamic lights, and it can’t do shadows. Lightprobes are a really neat solution for complex game worlds with static lighting.
Specialized shaders and detailed, high-contrast textures
The shaders in ShadowGun minimize per-pixel calculations and exploit complex and high-quality textures. See the Rendering Optimizations page for information on how to make textures that look great even when the shader is simple.
Cartoon Graphics
Who says your game has to look like a photo? If you make lighting and atmosphere the responsibility of the texture artist, not the engine, you hardly even have to worry about optimizing rendering.

What does not work:

Glow and other Post processing effects
Approximate such effects when possible by using blended **quads**, check out the Shadowgun project for an example of this.
Bumpmapping, especially with the built-in shaders
Use it sparingly, only on the most important characters or objects. Anything that can take up the whole screen probably shouldn't be bumpmapped.
Instead of using **bump maps**, bake more detail and contrast into the diffuse texture. The effect from League of Legends is an interesting example of this being used successfully in the industry.

But how do I actually do it?

See the Rendering Optimizations page.

Scripting and Gameplay Methods

Leave feedback

This section demonstrates ways that mobile developers write code and structure their games so that they run fast. The core idea here is that game design and optimization aren't really separate processes; decisions you make when you are designing your game can make it both fun and fast.

A historical example

You may remember old games where the player was only allowed one shot on the screen at a time, and reload speed was controlled by whether the bullet missed or not, instead of a timer. This technique is called **object pooling**, and it simplifies memory management, making programs run smoother.

The creators of space invaders only had a small amount of RAM, and they had to ensure that their program would never need to allocate more than was available. If they let the player fire once every second, and they offered a powerup that decreased the reload time to a half a second, they would have to ensure that there was enough memory space to allocate a lot of projectiles in the case where the player fires as fast as possible and all of the shots live for the longest possible time. That would probably pose a problem for them, so instead, they just allocated one projectile and left it at that. As soon as the projectile dies, it is simply deactivated, and repositioned and activated when it is fired again. But it always lives in the same space in memory and doesn't have to move around or be constantly deleted and recreated.

An optimization, or a gameplay gem?

This is hardly realistic, but it happens to be fun. Tension is released in a climactic moment when the alien invaders approach the ground, similar to a climax in film or literature. The invaders' close proximity gives the adept player near-instantaneous reload time, allowing them to miraculously defend earth by mashing the fire key in perfect time. Good game designs live in a bizarre space between the interactive narrative and the background technology that powers it all. It's hard to plan out awesome, fun, efficient stuff like this, because code logistics and user interaction are two wildly different and deeply finicky things, and using them together to synthesize something fresh and fun takes a lot of thought and experimentation.

You probably can't plan out every aspect of your game in terms of interaction and playing nice with mobile hardware simultaneously. It's more likely that these "gems" where the two meet in harmony will pop up as accidents while you're experimenting. But having a solid understanding of the way your code runs on the hardware you intend to deploy on will help. If you want to see the detailed technical explanation of why object pooling is better, and learn about memory allocation, see our Scripting Optimizations page.

Will X run fast on Mobiles?

Say you are beginning to work on a game, and you want to impress your players with lots of action and flashy stuff happening at once. How do you plan those things out? How do you know where the limits are, in game terms like how many coins, how many zombies, how many opponent cars, **etc**? It all depends on how you code your game.

Generally, if you write your game code the easy way, or the most general and versatile way, you will run into performance issues a lot sooner. The more you rely on specific structures and tricks to run your game, the more horizons will expand, and you will be able to cram more stuff on screen.

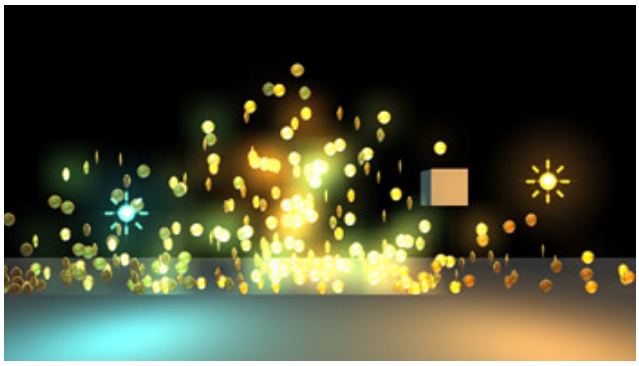
Easy and versatile, but slow

Rigidbodies limited to 2 dimensions in a 2D game.
Rigidbodies on projectiles.
Using Instantiate and Destroy a lot.
Lots of individual **3D objects** for collectables or characters.
Performing calculations every frame.
Using OnGUI for your GUI or HUD.

Complicated and limited, but faster

Writing your own physics code for a 2D game.
Handling **collision detection** for projectiles yourself.
Using Object Pooling instead of Instantiate and Destroy.
Using animated **sprites** on particles to represent simple objects.
Performing expensive calculations every few frames and caching the results.
A custom GUI solution.

Examples



Hundreds of rotating, dynamically lit, collectable coins onscreen at once

NO: Each coin is a separate object with a rigidbody and a script that rotates it and allows it to be picked up.
YES: The coins are a **particle system** with an animated texture, one script does the **collision** testing for all the coins and sets their color according to distance from a light.
This example is implemented in the Scripting Optimization page.

Your custom-built soft-body simulation

NO: The world has pillows lying around everywhere, which you can throw around and make piles of.
YES: Your character is a pillow, there is only one of them, and the situations it will be in are somewhat predictable (It only collides with spheres and axis-aligned cubes). You can probably code something which isn't a full-featured softbody simulation, but looks really impressive and runs fast.

30 enemy characters shooting at the player at once

NO: Each enemy has its own skinned **mesh**, a separate object for their weapon, and instantiates a rigidbody-based projectile every time they fire. Each enemy takes the state of all of its compatriots into account in a complicated AI script that runs every frame.

YES: Most of the enemies are far away, and are represented by single sprites, or, the enemies are 2D and are just a couple sprites anyway. Every enemy bullet is drawn by the same particle system and simulated by a script which does only rudimentary physics. Each enemy updates its AI state twice per second according to the state of the other enemies in its sector.

The how and why of script optimization

See the page on Optimizing Scripts.

Rendering Optimizations

Leave feedback

This section introduces the technicalities of **rendering** optimization. It shows how to bake lighting results for better performance, and how the developers of Shadowgun levered high-contrast textures, with lighting baked-in, to make their game look great. If you are looking for general information on what a mobile-optimized game looks like, check out the Graphics Methods page.

Get Artsy!

Sometimes optimizing the rendering in your game requires some dirty work. All of the structure that Unity provides makes it easy to get something working fast, but if you require top notch fidelity on limited hardware, doing things yourself and sidestepping that structure is the way to go, provided that you can introduce a key structural change that makes things a lot faster. Your tools of choice are editor **scripts**, simple **shaders**, and good old-fashioned art production.

How to Dive Under the Hood

First of all, check out this introduction to how shaders are written.

Built in shaders
Examine the source code of the built in shaders. Often, if you want to make a new shader that does something different, you can achieve it by taking parts of two already-existing shaders and putting them together.
Surface Shader Debugging (`#pragma debug`)
A CG Shader is generated from every surface shader, and then fully compiled from there. If you add `#pragma debug` to the top of your surface shader, when you open the compiled shader via the **inspector**, you can see the intermediate CG code. This is useful for inspecting how a specific part of a shader is actually calculated, and it can also be useful for grabbing certain aspects you want from a surface shader and applying them to a CG shader.
Shader Include Files
A lot of shader helper code is included in every shader, and usually it isn't used, but this is why you will sometimes see shaders calling functions like WorldReflectionVector which don't seem to be defined anywhere. Unity may has several built-in shader include files that contain these helper definitions. To find a specific function, you will need to search through all of the different includes.
These files are a major part of internal structure that Unity uses to make it easy to write shaders; the files provide things like real time shadows, different light types, **lightmaps**, and multiple platform support.
Hardware documentation
Take your time to study Apple documentation on best practices for writing shaders. Note that we would suggest to be more aggressive with floating point precision hints however.

Shadowgun in-depth

Shadowgun is a great graphical achievement considering the hardware it runs on. While the art quality seems to be the key to the puzzle, there are a couple tricks to achieving such quality that programmers can pull off to maximize their artists' potential.

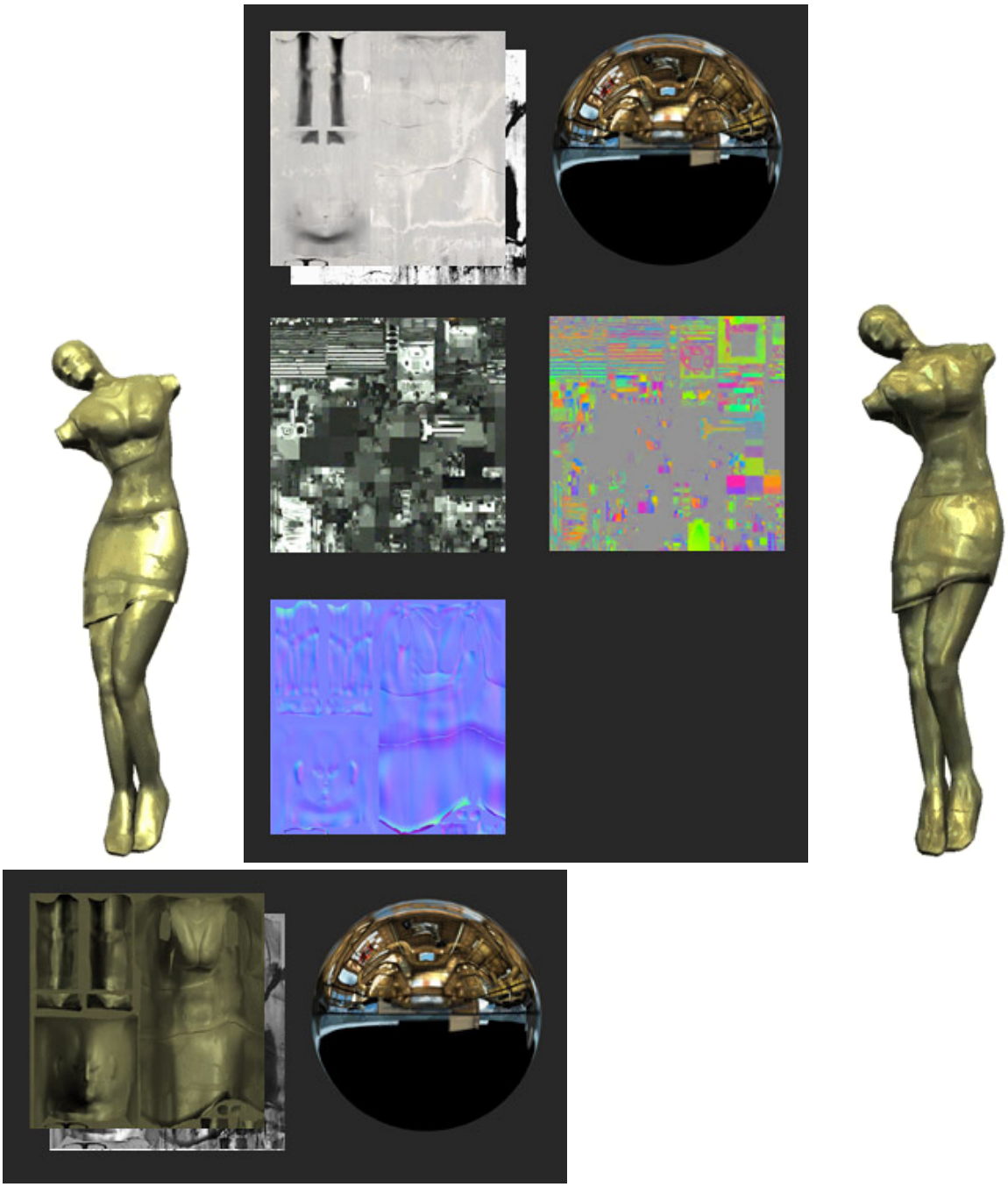
In the Graphics Methods page we used the golden statue in Shadowgun as an example of a great optimization; instead of using a normal map to give their statue some solid definition, they just baked lighting detail into the texture. Here, we will show you how and why you should use a similar technique in your own game.

Shader code for Real-Time vs Baked Golden Statue

```
// This is the pixel shader code for drawing normal-mapped
// specular highlights on static lightmapped geometry

// 5 texture reads, lots of instructions
```

```
SurfaceOutput o;

fixed4 tex = tex2D(_MainTex, IN.uv_MainTex);
fixed4 c = tex * _Color;
o.Albedo = c.rgb;

o.Gloss = tex.a;
o.Specular = _Shininess;

o.Normal = UnpackNormal(tex2D(_BumpMap, IN.uv_BumpMap));

float3 worldRefl = WorldReflectionVector (IN, o.Normal);
fixed4 reflcol = texCUBE (_Cube, worldRefl);
reflcol *= tex.a;
o.Emission = reflcol.rgb * _ReflectColor.rgb;
o.Alpha = reflcol.a * _ReflectColor.a;

fixed atten = LIGHT_ATTENUATION(IN);
fixed4 c = 0;

half3 specColor;
fixed4 lmtex = tex2D(unity_Lightmap, IN.lmap.xy);
fixed4 lmIndTex = tex2D(unity_LightmapInd, IN.lmap.xy);

const float3x3 unity_DirBasis = float3x3(
float3( 0.81649658,  0.0, 0.57735028),
float3(-0.40824830,  0.70710679, 0.57735027),
float3(-0.40824829, -0.70710678, 0.57735026) );

half3 lm = DecodeLightmap (lmtex);

half3 scalePerBasisVector = DecodeLightmap (lmIndTex);

half3 normalInRnmBasis = saturate (mul (unity_DirBasis, o.Normal));
lm *= dot (normalInRnmBasis, scalePerBasisVector);

return half4(lm, 1);
```

```
// This is the pixel shader code for lighting which is
// baked into the texture

// 2 texture reads, very few instructions

fixed4 c = tex2D (_MainTex, i.uv.xy);

c.xyz += texCUBE(_EnvTex,i.refl) * _ReflectionColor * c.a;

return c;
```

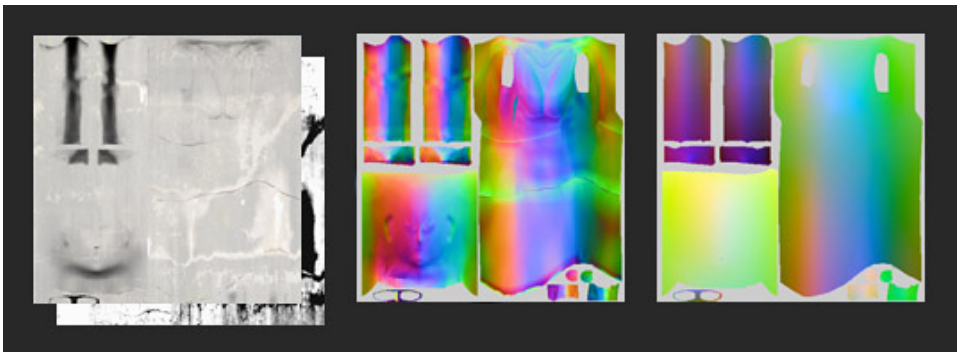


Render to Texel

The **real-time light** is definitely higher quality, but the performance gain from the baked version is massive. So how was this done? An editor tool called Render to Texel was created for this purpose. It bakes the light into the texture through the following process:

Transform the tangent space normal map to world space via script.
Create a world space position map via script.
Render to Texture a fullscreen pass of a the entire texture using the two previous maps, with one additional pass per light.
Average results from several different vantage points to yield something which looks plausible all around, or at least from common viewing angles in your game.

This is how the best graphics optimizations work. They sidestep tons of calculations by preforming them in the editor or before the game runs. In general, this is what you want to do:



Create something that looks great, don't worry about performance.
Use tools like Unity's **lightmapper** and editor extensions like Render to Texel and **Sprite Packer** to bake it down to something which is very simple to render.
Making your own tools is the best way to do this, you can create the perfect tool suited for whatever problem your game presents.
Create shaders and scripts which modulate your baked output to give it some sort of "shine"; an eye-catching effect to create an illusion of dynamic light.

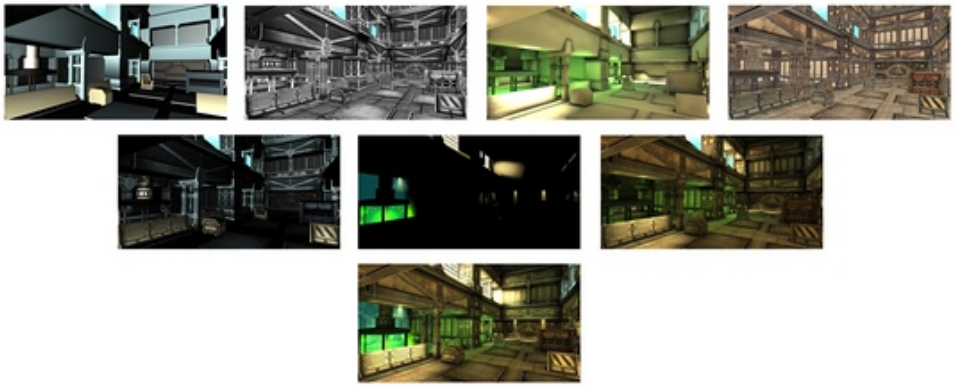
Concept of Light Frequency

Just like the Bass and Treble of an audio track, images also have high-frequency and low-frequency components, and when you're rendering, it's best to handle them in different ways, similar to how stereos use subwoofers and tweeters to produce a full body of sound. One way to visualize the different frequencies of an image is to use the "High Pass" filter in Photoshop. **Filters->Other->High Pass**. If you have done audio work before, you will recognize the name High Pass. Essentially what it does is cut off all frequencies lower than X, the parameter you pass to the filter. For images, Gaussian Blur is the equivalent of a Low Pass.

This applies to realtime graphics because frequency is a good way to separate things out and determine how to handle what. For example, in a basic lightmapped environment, the final image is obtained by composite of the lightmap, which is low frequency, and the textures, which are high-frequency. In Shadowgun, low frequency light is applied to characters quickly with **light probes**, high frequency light is faked through the use of a simple bumpmapped shader with an arbitrary light direction.

In general, by using different methods to render different frequencies of light, for example, baked vs dynamic, per-object vs per-level, per **pixel** vs per-vertex, **etc**, you can create full bodied images on limited hardware. Stylistic choices aside, it's generally a good idea to try to have strong variation colors or values at both high and low frequencies.

Frequency in Practice: Shadowgun Decomposition



Top Row
Ultra-Low-Frequency Specular Vertex Light (Dynamic) | High Frequency Alpha Channel | Low Frequency Lightmap | High Frequency Albedo
Mid Row
Specular Vertex Light * Alpha | High Frequency Additive Details | Lightmap * Color Channel
Bottom
Final Sum

Note: Usually these decompositions refer to steps in a deferred renderer, but that's not the case here. All of this is done in just one pass. These are the two relevant shaders which this composition was based on:

Lightmapped with Virtual Gloss Per-Vertex Additive

```
Shader "MADFINGER/Environment/Virtual Gloss Per-Vertex Additive (Supports Lightmap)" {
Properties {
    _MainTex ("Base (RGB) Gloss (A)", 2D) = "white" {}
    //_MainTexMipBias ("Base Sharpness", Range (-10, 10)) = 0.0
    _SpecOffset ("Specular Offset from Camera", Vector) = (1, 10, 2, 0)
    _SpecRange ("Specular Range", Float) = 20
    _SpecColor ("Specular Color", Color) = (0.5, 0.5, 0.5, 1)
    _Shininess ("Shininess", Range (0.01, 1)) = 0.078125
    _ScrollingSpeed("Scrolling speed", Vector) = (0,0,0,0)
}

SubShader {
    Tags { "RenderType"="Opaque" "LightMode"="ForwardBase"}
    LOD 100



    CGINCLUDE
    #include "UnityCG.cginc"
    sampler2D _MainTex;
    float4 _MainTex_ST;
    samplerCUBE _ReflTex;

    #ifdef LIGHTMAP_ON
    float4 unity_LightmapST;
    sampler2D unity_Lightmap;
    #endif

    //float _MainTexMipBias;
```

```
    float3 _SpecOffset;
    float _SpecRange;
    float3 _SpecColor;
    float _Shininess;
    float4 _ScrollingSpeed;

    struct v2f {
        float4 pos : SV_POSITION;
        float2 uv : TEXCOORD0;
        #ifdef LIGHTMAP_ON
        float2 lmap : TEXCOORD1;
        #endif
        fixed3 spec : TEXCOORD2;
    };


    v2f vert (appdata_full v)
    {
        v2f o;
        o.pos = UnityObjectToClipPos(v.vertex);

        o.uv = v.texcoord + frac(_ScrollingSpeed * _Time.y);

        float3 viewNormal = UnityObjectToViewPos(v.normal);
        float3 viewPos = UnityObjectToViewPos(v.vertex);
        float3 viewDir = float3(0,0,1);
        float3 viewLightPos = _SpecOffset * float3(1,1,-1);

        float3 dirToLight = viewPos - viewLightPos;

        float3 h = (viewDir + normalize(-dirToLight)) * 0.5;
        float atten = 1.0 - saturate(length(dirToLight) / _SpecRange);

        o.spec = _SpecColor * pow(saturate(dot(viewNormal, normalize(h))), _Shininess * 12

        #ifdef LIGHTMAP_ON
        o.lmap = v.texcoord1.xy * unity_LightmapST.xy + unity_LightmapST.zw;
        #endif
        return o;
    }
    ENDCG


    Pass {
        CGPROGRAM
        #pragma vertex vert
        #pragma fragment frag
        fixed4 frag (v2f i) : SV_Target
        {
            fixed4 c = tex2D (_MainTex, i.uv);

            fixed3 spec = i.spec.rgb * c.a;
```

```
            #if 1
            c.rgb += spec;
            #else
            c.rgb = c.rgb + spec - c.rgb * spec;
            #endif

            #ifdef LIGHTMAP_ON
            fixed3 lm = DecodeLightmap (tex2D(unity_Lightmap, i.lmap));
            c.rgb *= lm;
            #endif

            return c;
        }
        ENDCG
    }
}
}
```

Lightprobes with Virtual Gloss Per-Vertex Additive

```
Shader "MADFINGER/Environment/Lightprobes with VirtualGloss Per-Vertex Additive" {
Properties {
    _MainTex ("Base (RGB) Gloss (A)", 2D) = "white" {}
    _SpecOffset ("Specular Offset from Camera", Vector) = (1, 10, 2, 0)
    _SpecRange ("Specular Range", Float) = 20
    _SpecColor ("Specular Color", Color) = (1, 1, 1, 1)
    _Shininess ("Shininess", Range (0.01, 1)) = 0.078125
    _SHLightingScale("LightProbe influence scale",float) = 1
}

SubShader {
    Tags { "RenderType"="Opaque" "LightMode"="ForwardBase"}
    LOD 100



    CGINCLUDE
    #pragma multi_compile _ LIGHTMAP_ON
    #include "UnityCG.cginc"
    sampler2D _MainTex;
    float4 _MainTex_ST;


    float3 _SpecOffset;
    float _SpecRange;
    float3 _SpecColor;
    float _Shininess;
```

```
float _SHLightingScale;

struct v2f {
    float4 pos : SV_POSITION;
    float2 uv : TEXCOORD0;
    float3 refl : TEXCOORD1;
    fixed3 spec : TEXCOORD3;
    fixed3 SHLighting: TEXCOORD4;
};


v2f vert (appdata_full v)
{
    v2f o;
    o.pos = UnityObjectToClipPos(v.vertex);
    o.uv = v.texcoord;

    float3 worldNormal = UnityObjectToWorldDir(v.normal);
    float3 viewNormal = UnityObjectToViewPos(v.normal);
    float4 viewPos = UnityObjectToViewPos(v.vertex);
    float3 viewDir = float3(0,0,1);
    float3 viewLightPos = _SpecOffset * float3(1,1,-1);

    float3 dirToLight = viewPos.xyz - viewLightPos;

    float3 h = (viewDir + normalize(-dirToLight)) * 0.5;
    float atten = 1.0 - saturate(length(dirToLight) / _SpecRange);

    o.spec = _SpecColor * pow(saturate(dot(viewNormal, normalize(h))), _Shininess * 12

    o.SHLighting    = ShadeSH9(float4(worldNormal,1)) * _SHLightingScale;

    return o;
}
ENDCG


Pass {
    CGPROGRAM
    #pragma vertex vert
    #pragma fragment frag
    fixed4 frag (v2f i) : SV_Target
    {
        fixed4 c    = tex2D (_MainTex, i.uv);

        c.rgb *= i.SHLighting;
        c.rgb += i.spec.rgb * c.a;

        return c;
    }
    ENDCG
}
```

```
}
}
```

Best Practices

GPU optimization: Alpha-Testing

Some GPUs, particularly ones found in mobile devices, incur a high performance overhead for alpha-testing (or use of the **discard** and **clip** operations in pixel shaders). You should replace alpha-test shaders with alpha-blended ones if possible. Where alpha-testing cannot be avoided, you should keep the overall number of visible alpha-tested pixels to a minimum.

iOS Texture Compression

Some images, especially if using **iOS**/Android PVR texture **compression**, are prone to visual artifacts in the alpha channel. In such cases, you might need to tweak the PVRT **compression** parameters directly in your imaging software. You can do that by installing the **PVR export plugin** or using PVRTexTool from Imagination Tech, the creators of the **PVRTC** format. The resulting compressed image file with a **.pvr** extension will be imported by the Unity editor directly and the specified compression parameters will be preserved. If PVRT-compressed textures do not give good enough visual quality or you need especially crisp imaging (as you might for GUI textures) then you should consider using 16-bit textures instead of 32-bit. By doing so, you will reduce the memory bandwidth and storage requirements by half.

Android Texture Compression

All Android devices with support for OpenGL ES 2.0 also support the ETC1 compression format; it's therefore encouraged to whenever possible use ETC1 as the prefered **texture format**.

If targeting a specific graphics architecture, such as the Nvidia Tegra or Qualcomm Snapdragon, it may be worth considering using the proprietary compression formats available on those architectures. The Android Market also allows filtering based on supported **texture compression** format, meaning a distribution archive (.apk) with for example DXT compressed textures can be prevented for download on a device which doesn't support it.

An Exercise

Download Render to Texel. Bake lighting on your model. Run the High Pass filter on the result in Photoshop. Edit the "Mobile/Cubemapped" shader, included in the Render to Texel package, so that the missing low-frequency light details are replaced by vertex light.

Optimizing Scripts

Leave feedback

This section demonstrates how you would go about optimizing the actual **scripts** and methods your game uses, and it also goes into detail about the reasons why the optimizations work, and why applying them will benefit you in certain situations.

Profiler is King

There is no such thing as a list of boxes to check that will ensure your project runs smoothly. To optimize a slow project, you have to profile to find specific offenders that take up a disproportionate amount of time. Trying to optimize without profiling or without thoroughly understanding the results that the profiler gives is like trying to optimize with a blindfold on.

Internal mobile profiler

You can use the internal profiler to figure out what kind of process is slowing your game down, be it physics, scripts, or **rendering**, but you can't drill down into specific scripts and methods to find the actual offenders. However, by building switches into your game which enable and disable certain functionality, you can narrow down the worst offenders significantly. For example, if you remove the enemy characters' AI script and the framerate doubles, you know that the script, or something that it brings into the game, has to be optimized. The only problem is that you may have to try a lot of different things before you find the problem.

For more about profiling on mobile devices, see the profiling section.

Optimized by Design

Attempting to develop something which is fast from the beginning is risky, because there is a trade-off between wasting time making things that would be just as fast if they weren't optimized and making things which will have to be cut or replaced later because they are too slow. It takes intuition and knowledge of the hardware to make good decisions in this regard, especially because every game is different and what might be a crucial optimization for one game may be a flop in another.

Object Pooling

We gave object pooling as an example of the intersection between good gameplay and good code design in the introduction to optimized scripting methods. Using object pooling for ephemeral objects is faster than creating and destroying them, because it makes memory allocation simpler and removes dynamic memory allocation overhead and Garbage Collection, or GC.

Memory Allocation

Simple Explanation of what Automatic Memory Management is

Scripts you write in Unity use automatic memory management. Just about all scripting languages do this. In contrast, lower level languages such as C and C++ use manual memory allocation, where the programmer is allowed to read and write from memory addresses directly, and as a consequence they are responsible for removing every object they create. For example, if you create objects in your C++, you have to manually de-allocate the memory that they take up when you are done with them. In a scripting language, it is enough to say `objectReference = null;`

Note: If I have a game object variable like `GameObject myGameObject;` or `var myGameObject : __GameObject__;`, why isn't it destroyed when I say `myGameObject = null;`?

The game object is still referenced by Unity, because Unity has to maintain a reference to it in order for it to be drawn, updated, **etc**. Calling `Destroy(myGameObject);` removes that reference and deletes the object.
But if you create an object that Unity has no idea about, for example, an instance of a class that does not inherit from anything (in contrast, most classes or "script components" inherit from MonoBehaviour) and then set your reference variable to it to null, what actually happens is that the object is lost as far as your script and Unity are concerned; they can't access it and will never see it again, but it stays in memory. Then, some time later, the Garbage Collector runs, and it removes anything in memory that is not referenced anywhere. It is able to do this because, behind the **scenes**, the number of references to each block of memory is kept track of. This is one reason why scripting languages are slower than C++.

Read more about Automatic Memory Management and the Garbage Collector.

How to Avoid Allocating Memory

Every time an object is created, memory is allocated. Very often in code, you are creating objects without even knowing it.

`Debug.Log("boo" + "hoo");` creates an object.
Use `System.String.Empty` instead of `""` when dealing with lots of strings.
Immediate Mode GUI (UnityGUI) is slow and should not be used at any time when performance is an issue.
Difference between class and struct:
Classes are objects and behave as references. If **Foo** is a class and

```
Foo foo = new Foo();
MyFunction(foo);
```

then **MyFunction** will receive a reference to the original Foo object that was allocated on the heap. Any changes to **foo** inside **MyFunction** will be visible anywhere foo is referenced.

Classes are data and behave as such. If **Foo** is a struct and

```
Foo foo = new Foo();
MyFunction(foo);
```

then **MyFunction** will receive a copy of **foo**. **foo** is never allocated on the heap and never garbage collected. If **MyFunction** modifies it's copy of **foo**, the other **foo** is unaffected.

Objects which stick around for a long time should be classes, and objects which are ephemeral should be structs.
Vector3 is probably the most famous struct. If it were a class, everything would be a lot slower.

Why Object Pooling is Faster

The upshot of this is that **using Instantiate and Destroy a lot gives the Garbage Collector a lot to do**, and this can cause a "hitch" in gameplay. As the Automatic Memory Management page explains, there are other ways to get around the common performance hitches that surround Instantiate and Destroy, such as triggering the Garbage Collector manually when nothing is going on, or triggering it very often so that a large backlog of unused memory never builds up.

Another reason is that, when a specific **prefab** is instantiated for the first time, sometimes additional things have to be loaded into RAM, or textures and meshes need to be uploaded to the GPU. This can cause a hitch as well, and with object pooling, this happens when the level loads instead of during gameplay.

Imagine a puppeteer who has an infinite box of puppets, where every time the script calls for a character to appear, they get a new copy of its puppet out of the box, and every time the character exits the stage, they toss the current copy. Object pooling is the equivalent of getting all the puppets out of the box before the show starts, and leaving them on the table behind the stage whenever they are not supposed to be visible.

Why Object Pooling can be Slower

One issue is that the creation of a pool reduces the amount of heap memory available for other purposes; so if you keep allocating memory on top of the pools you just created, you might trigger garbage collection even more often. Not only that, every collection will be slower, because the time taken for a collection increases with the number of live objects. With these issues in mind, it should be apparent that performance will suffer if you allocate pools that are too large or keep them active when the objects they contain will not be needed for some time. Furthermore, many types of objects don't lend themselves well to object pooling. For example, the game may include spell effects that persist for a considerable time or enemies that appear in large numbers but which are only killed gradually as the game progresses. In such cases, the performance overhead of an object pool greatly outweighs the benefits and so it should not be used.

Implementation

Here's a simple side by side comparison of a script for a simple projectile, one using Instantiation, and one using Object Pooling.

```
// GunWithInstantiate.js

#pragma strict

var prefab : ProjectileWithInstantiate;

var power = 10.0;

function Update () {
    if(Input.GetButtonDown("Fire1")) {
        var instance : ProjectileWithInstantiate =
            Instantiate(prefab, transform.position, transform.rotation);
        instance.velocity = transform.forward * power;
    }
}
```

```
// GunWithObjectPoo

#pragma strict

var prefab : Projec
var maximumInstance
var power = 10.0;

private var instanc

static var stackPos

function Start ()
    instances = new
    for(var i = 0;
        // place th
        instances[i
        // disable
        instances[i
    }
}

function Update ()
    if(Input.GetBut
        var instanc
        if(instance
            instanc
        }
    }
}

function GetNextAva
    for(var i = 0;
        if(!instanc
    }
    return null;
}
```

```
// ProjectileWithInstantiate.js

#pragma strict

var gravity = 10.0;
var drag = 0.01;
var lifetime = 10.0;
```

```
// ProjectileWithOb

#pragma strict

var gravity = 10.0
var drag = 0.01;
var lifetime = 10.0
```

```
var velocity : Vector3;

private var timer = 0.0;





function Update () {
    velocity -= velocity * drag * Time.deltaTime;
    velocity -= Vector3.up * gravity * Time.deltaTime;
    transform.position += velocity * Time.deltaTime;

    timer += Time.deltaTime;
    if(timer > lifetime) {

        Destroy(gameObject);
    }
}
```

```
var velocity : Vect

private var timer =

function Initialize
    transform.posit
    transform.rotat
    velocity = pare
    timer = 0;
    enabled = true,
}

function Update ()
    velocity -= vel
    velocity -= Vec
    transform.posit

    timer += Time.
    if(timer > life
        transform.p
        enabled = 
    }
}
```

Of course, for a large, complicated game, you will want to make a generic solution that works for all your prefabs.

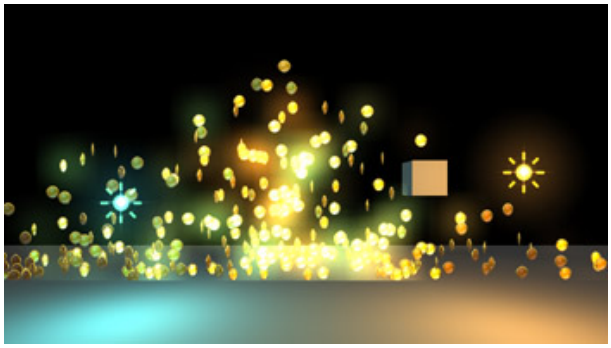
Another Example: Coin Party!

The example of "Hundreds of rotating, dynamically lit, collectable coins onscreen at once" which was given in the Scripting Methods section will be used to demonstrate how script code, Unity **components** like the **Particle System**, and custom **shaders** can be used to create a stunning effect without taxing the weak mobile hardware.

Imagine that this effect lives in the context of a 2D sidescrolling game with tons of coins that fall, bounce, and rotate. The coins are dynamically lit by point lights. We want to capture the light glinting off the coins to make our game more impressive.

If we had powerful hardware, we could use a standard approach to this problem. Make every coin an object, shade the object with either vertex-lit, forward, or deferred lighting, and then add glow on top as an image effect to get the brightly reflecting coins to bleed light onto the surrounding area.

But mobile hardware would choke on that many objects, and a glow effect is totally out of the question. So what do we do?

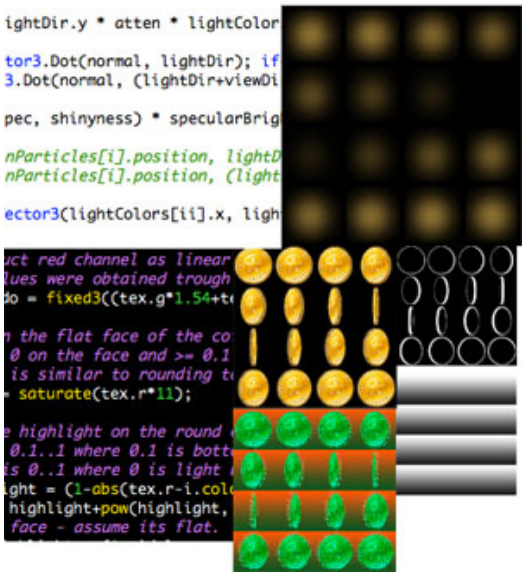


Animated Sprite Particle System

If you want to display a lot of objects which all move in a similar way and can never be carefully inspected by the player, you might be able to render large amounts of them in no time using a particle system. Here are a few stereotypical applications of this technique:

Collectables or Coins
Flying Debris
Hordes or Flocks of Simple Enemies
Cheering Crowds
Hundreds of Projectiles or Explosions

There is a free editor extension called Sprite Packer that facilitates the creation of animated **sprite** particle systems. It renders frames of your object to a texture, which can then be used as an animated **sprite** sheet on a particle system. For our use case, we would use it on our rotating coin.



Reference Implementation

Included in the Sprite Packer project is an example that demonstrates a solution to this exact problem.

It uses a family of **assets** of all different kinds to achieve a dazzling effect on a low computing budget:

A control script
Specialized textures created from the output of the SpritePacker
A specialized shader which is intimately connected with both the control script and the texture.

A readme file is included with the example which attempts to explain why and how the system works, outlining the process that was used to determine what features were needed and how they were implemented. This is that file:

The problem was defined as “Hundreds of rotating, dynamically lit, collectable coins onscreen at once.”

The naive approach is to Instantiate a bunch of copies of a coin prefab, but instead we are going to use particles to render our coins. However, this introduces a number of challenges that we have to overcome.

Viewing angles are a problem because particles don’t have them.
We assume that the **camera** stays right-side up and the coins rotate around the Y-axis.
We create the illusion of coin rotation with an animated texture that we packed using the SpritePacker.
This introduces a new problem: Monotony of rotating coins all rotating at the same speed and in the same direction
We keep track of rotation and lifetime ourselves and “render” rotation to the particle lifetimes in script to fix this.
Normals are a problem because particles don’t have them, and we need real time lighting.
Generate a single normal vector for the face of the coin in each animation frame generated by the Sprite Packer.
Do Blinn-Phong lighting for each particle in script, based on the normal vector grabbed from the above list.

Apply the result to the particle as a color.
Handle the face of the coin and the rim of the coin separately in the shader. Introduces a new problem: How does the shader know where the rim is, and what part of the rim it's on?
Can't use UV's, they are already used for the animation.
Use a texture map.
Need Y-position relative to coin.
Need binary "on face" vs "on rim".
We don't want to introduce another texture, more texture reads, more texture memory.
Combine needed information into one channel and replace one of the texture's color channels with it.
Now our coin is the wrong color! What do we do?
Use the shader to reconstruct missing channel as a combination of the two remaining channels.
Say we want glow from light glinting off our coins. Post process is too expensive for mobile devices.
Create another particle system and give it a softened, glowy version of the coin animation.
Color a glow only when the corresponding coin's color is super bright.
Can't have glow rendered on every coin every frame - fill rate killer.
Reset glows every frame, only position ones with brightness > 0.
Physics is a problem, collecting coins is a problem - particles don't collide very well.
Could use built-in particle **collision**?
Instead, just wrote collision into the script.
Finally, we have one more problem - this script does a lot, and its getting slow!
Performance scales linearly with number of active coins.
Limit maximum coins. This works well enough to acheive our goal: 100 coins, 2 lights, runs really fast on mobile devices.
Things to try to optimize further:
Instead of calculating lighting for every coin individually, cut the world into chunks and calculate lighting conditions for every rotation frame in every chunk.
Use as a lookup table with coin position and coin rotation as indices.
Increase fidelity by using bilinear **interpolation** with position.
Sparse updates on the lookup table, or, entirely static lookup table.
Use **Light Probes** for this? *Instead of calculating lighting in script, use normal-mapped particles?
Use "Display Normals" shader to bake frame animation of normals.
Limits number of lights.
Fixes slow script problem.

The end goal of this example or "moral of the story" is that if there is something which your game really needs, and it causes lag when you try to achieve it through conventional means, that doesn't mean that it is impossible, it just means that you have to put in some work on a system of your own that runs much faster.

Techniques for Managing Thousands of Objects

These are specific scripting optimizations which are applicable in situations where hundreds or thousands of dynamic objects are involved. Applying these techniques to every script in your game is a terrible idea; they should be reserved as tools and design guidelines for large scripts which handle tons of objects or data at run time.

Avoid or minimize O(n2) operations on large data sets

In computer science, the Order of an operation, denoted by O(n), refers to the way that the number of times that the operation has to be evaluated increases as the number of objects it is applied to (n) increases.

For example, consider a basic sorting algorithm. I have **n** numbers and I want to sort them from smallest to largest.

```
void sort(int[] arr) {
   int i, j, newValue;
   for (i = 1; i < arr.Length; i++) {
       // record
       newValue = arr[i];
```

```
        //shift everything that is larger to the right
        j = i;
        while (j > 0 && arr[j - 1] > newValue) {
            arr[j] = arr[j - 1];
            j--;
        }
        // place recorded value to the left of large values
        arr[j] = newValue;
    }
}
```

The important part is that there are two loops here, one inside the other.

```
for (i = 1; i < arr.Length; i++) {
    ...
    j = i;
    while (j > 0 && arr[j - 1] > newValue) {
        ...
        j--;
    }
}
```

Let's assume that we give the algorithm the worst possible case: the input numbers are sorted, but in reverse order. In that case, the innermost loop will run **j** times. On average, as **i** goes from **1** to **arr.Length–1**, **j** will be **arr.Length/2**. In terms of **O(n)**, **arr.Length** is our **n**, so, in total, the innermost loop runs ***n/2** times, or* $n^2/2$** times. But in **O(n)** terms, we chuck all constants like **1/2**, because we want to talk about the way that the number of operations increases, not the actual number of operations. So the algorithm is **O(n^2)**. The order of an operation matters a lot if the data set is large, because the number of operations can explode exponentially.

An in-game example of an **O(n^2)** operation is 100 enemies, where the AI of each enemy takes the movements of every other enemy into account. It might be faster to divide the map into cells, record the movement of each enemy into the nearest cell, and then have each enemy sample the nearest few cells. That would be an **O(n)** operation.

Cache references instead of performing unnecessary searches

Say you have 100 enemies in your game, and they all move towards the player.

```
// EnemyAI.js
var speed = 5.0;

function Update () {
    transform.LookAt(GameObject.FindWithTag("Player").transform);
    // this would be even worse:
    //transform.LookAt(FindObjectOfType(Player).transform);

    transform.position += transform.forward * speed * Time.deltaTime;
}
```

That could be slow, if there are enough of them running at the same time. Little known fact: all of the component accessors in MonoBehaviour, things like **transform**, **renderer**, and **audio**, are equivalent to their **GetComponent(Transform)** counterparts, and they are actually a bit slow. **GameObject.FindWithTag** has been optimized, but in some cases, for example, in inner loops, or on scripts that run on a lot of instances, this script might be a bit slow.

This is a better version of the script.

```
// EnemyAI.js
var speed = 5.0;

private var myTransform : Transform;
private var playerTransform : Transform;

function Start () {
   myTransform = transform;
   playerTransform = GameObject.FindWithTag("Player").transform;
}

function Update () {
   myTransform.LookAt(playerTransform);

   myTransform.position += myTransform.forward * speed * Time.deltaTime;
}
```

Minimize expensive math functions

Transcendental functions (**Mathf.Sin**, **Mathf.Pow**, etc), Division, and Square Root all take about 100x the time of a multiplication. (In the grand scheme of things, no time at all, but if you are calling them thousands of times per frame it can add up).

The most common case of this is vector normalization. If you are normalizing the same vector over and over, consider normalizing it once instead and caching the result for use later.

If you are both using the length of a vector and normalizing it, it would be faster to obtain the normalized vector by multiplying the vector by the reciprocal of the length rather than by using the **.normalized** property.

If you are comparing distances, you don't have to compare the actual distances. You can compare the squares of the distances instead by using the **.sqrMagnitude** property and save a square root or two.

Another one, if you are dividing over and over by a constant c, you can multiply by the reciprocal instead. Calculate the reciprocal first by doing **1.0/c**.

Only execute expensive operations occasionally, e.g. Physics.Raycast()

If you have to do something expensive, you might be able to optimize it by doing it less often and caching the result. For example, consider a projectile script that uses Raycast:

```
// Bullet.js
var speed = 5.0;

function FixedUpdate () {
   var distanceThisFrame = speed * Time.fixedDeltaTime;
   var hit : RaycastHit;
```

```
    // every frame, we cast a ray forward from where we are to where we will be next frame
    if(Physics.Raycast(transform.position, transform.forward, hit, distanceThisFrame)) {
        // Do hit
    } else {
        transform.position += transform.forward * distanceThisFrame;
    }
}
```

Right away, we could improve the script by replacing FixedUpdate with Update and fixedDeltaTime with deltaTime. FixedUpdate refers to the Physics update, which happens more often than the frame update. But let's go even further by only raycasting every n seconds. A smaller n gives greater temporal resolution, and a bigger n gives better performance. The bigger and slower your targets are, the bigger n can be before temporal aliasing occurs. (Appearance of latency, where the player hit the target, but the explosion appears where the target used to be n seconds ago, or the player hit the target, but the projectile goes right through).

```
// BulletOptimized.js
var speed = 5.0;
var interval = 0.4; // this is 'n', in seconds.

private var begin : Vector3;
private var timer = 0.0;
private var hasHit = false;
private var timeTillImpact = 0.0;
private var hit : RaycastHit;

// set up initial interval
function Start () {
   begin = transform.position;
   timer = interval+1;
}

function Update () {
   // don't allow an interval smaller than the frame.
   var usedInterval = interval;
   if(Time.deltaTime > usedInterval) usedInterval = Time.deltaTime;

   // every interval, we cast a ray forward from where we were at the start of this interval
   // to where we will be at the start of the next interval
   if(!hasHit && timer >= usedInterval) {
       timer = 0;
       var distanceThisInterval = speed * usedInterval;

       if(Physics.Raycast(begin, transform.forward, hit, distanceThisInterval)) {
           hasHit = true;
           if(speed != 0) timeTillImpact = hit.distance / speed;
       }

       begin += transform.forward * distanceThisInterval;
   }

   timer += Time.deltaTime;
```

```
    // after the Raycast hit something, wait until the bullet has traveled
    // about as far as the ray traveled to do the actual hit
    if(hasHit && timer > timeTillImpact) {
        // Do hit
    } else {
        transform.position += transform.forward * speed * Time.deltaTime;
    }
}
```

Minimize callstack overhead in inner loops

Just calling a function has a little bit of overhead in itself. If you are calling things like **x = Mathf.Abs(x)** thousands of times per frame, it might be better to just do **x = (x > 0 ? x : -x);** instead.

Optimizing Physics Performance

The NVIDIA PhysX **physics engine** used by Unity is available on mobiles, but the performance limits of the hardware will be reached more easily on mobile platforms than desktops.

Here are some tips for tuning physics to get better performance on mobiles:-

You can adjust the **Fixed Timestep** setting (in the Time manager) to reduce the time spent on physics updates. Increasing the timestep will reduce the CPU overhead at the expense of the accuracy of the physics. Often, lower accuracy is an acceptable tradeoff for increased speed.
Set the **Maximum Allowed Timestep** in the Time manager in the 8–10fps range to cap the time spent on physics in the worst case scenario.
Mesh colliders have a much higher performance overhead than primitive colliders, so use them sparingly. It is often possible to approximate the shape of a **mesh** by using child objects with primitive colliders. The child colliders will be controlled collectively as a single compound **collider** by the **rigidbody** on the parent.
While **wheel colliders** are not strictly colliders in the sense of solid objects, they nonetheless have a high CPU overhead.

Experimental

Leave feedback

Unity features which are in development.

Unity Technologies supplies some experimental features with the standard Unity Editor installer, which you download from the Unity store. Other experimental features you need to download from the Unity Technologies GitHub and install separately, once you have installed the Unity Editor.

In-Editor experimental features

To access experimental features in Unity, go to **Window** > **Experimental** > [name of feature]

Experimental features:

Look Dev: An image-based lighting tool that allows you to check and compare different **Assets** through a viewer to ensure they are correctly authored for various lighting conditions.

Plugin experimental features

To access Unity experimental feature plugins, download them from the Unity Technologies GitHub and install them separately to the Unity Editor.

2017–06–07 Page amended with editorial review

GitHub link and plugins added in documentation version 5.6

Getting started with UIElements

Leave feedback

UIElements is a new experimental feature introduced in Unity 2017.1.

It is recommended that you look at the example project as you read through this document.

The example project is located at https://github.com/Unity-Technologies/UIElementsExamples.

Disclaimer: *this experimental feature is incomplete and subject to API changes. While we appreciate any feedback, please remember that UIElements is still in active development.

Also, please note that changes introduced in 2017.2 won't be backported to 2017.1.*

Introduction

So far the Unity editor user interface has been mostly built around the immediate mode **UI** system. While IMGUI shines in some contexts it has some serious design limitations which are affecting the productivity of everyone working on Editor features.

This was our motivation to start working on UIElements, a retained-mode UI system that opens the door to improved performance and new features such as stylesheets, dynamic/contextual event handling, accessibility and much more.

Visual tree

The visual tree is an object graph made of lightweight nodes called "visual elements". This is a graph that can be traversed for event processing, layout and painting. These nodes are allocated on the C# heap, either manually or via **asset** loading in the future.

VisualContainer and VisualElement

`VisualElement` is the common base class for all nodes of the visual tree. The `VisualElement` base class contains properties for styles, layout data, local transformations, event handlers, and so on.

`VisualElement` has several subclasses, including `VisualContainer` and specialized controls, which define additional behaviour or functionality. However, it is not strictly required to derive from any class to work with UIElements, as built-in classes should contain most of the required functionality.

A `VisualContainer` object contains `VisualElement` child objects. The `VisualContainer` class has several methods to work with its list of children.

Note: UIElements classes are currently inside of the `UnityEngine.Experimental.UIElements` namespace.

Connectivity

A newly created element is ignored until connected to the root object, called the **"panel"**. You usually add elements to an existing container object, such as EditorWindow.rootVisualContainer, to attach your UI to the panel.

Note: to prevent accidental usage of this property while UIElements is experimental, it's necessary to go through the extension method `GetRootVisualContainer()` from the `UnityEditor.Experimental.UIElements` namespace.

To test if a `VisualElement` is connected to a root, you can test for the `panel` property of this element (`null` if not connected).

Painting order

Elements of the tree are drawn in the following order:

parents are drawn before their children

siblings are painted in the order of their parent's children list

The only way to change painting order is thus to re-order `VisualElement` objects in their parents.

It is possible to draw a given subtree in a `RenderTexture` and re-use the **pixels** for later repaint events by enabling `VisualContainer.usePixelCaching`.

Note: It is possible to implement custom painting by subclassing `VisualElement` and overriding the DoRepaint() method, however we do not recommend this approach as it will change in future releases.

Position, transforms and coordinate systems

The layout system affects each VisualElement.position property (of type Rect). This rect is expressed as pixels in relation to the coordinate space of its parent. You can assign this position manually as shown in the first example script.

Each `VisualElement` also has a transform property (of type `Matrix4x4`) which transform its local coordinates space (before taking in account the position rectangle) from its parent space.

The final `VisualElement` window space coordinates can be retrieved via the `globalBound` property, taking account of transforms and positions of the full ancestry of a `VisualElement`.

Note: in 2017.2, we will replace `transform` by Translate, Rotate and Scale properties for VisualElement, similar to Unity's Transform **component**.

The `VisualElementExtensions` static class contains several extensions to perform transformations between coordinates system.

Layout

`VisualElement` and `VisualContainer` classes contain several properties to control the layout. By default all elements are part of the layout, with the following default behaviours:

Containers will distribute their children vertically

Container's position rectangles will encompass their children's rectangles (unless explicitly restricted by other layout properties)

VisualElement with text will measure their text and size themselves accordingly (unless explicitly restricted by other layout properties)

Default controls also come with pre-defined styles which will affect their layout

Layout in UIElements is relying on the Yoga open source project, which implements a subset of the HTML/CSS layout system called Flexbox.

Here are some resources to help you get started with Flexbox:

Yoga's own documentation : mapping of properties is almost entirely 1-to–1

CSS-Tricks guide to Flexbox : most properties are supported with some minor differences

However here is a quick layout cheat sheet:

In the most basic cases, you can simply assign `width` and `height` to elements in order to give them a size

Flexible size can be given via the `flex` property, which acts as weight of an element in regards to its siblings.

You can switch to horizontal layout by assigning the `flexDirection` property to Row

Relative positioning lets you offset an element based on its original layout position

Absolute positioning lets you place an element relative to its parent position rectangle, in which case it does not contribute affect the layout of its siblings or parent

Elements which have their "position" property assigned directly via the API will be automatically considered as "absolute"

To prevent elements from stretching in the cross-axis, assign `FlexStart` to the `alignSelf` property of each `VisualElement` child object or the parent's `alignItems` property

Styles

```
    margin-right:8;
    background-color: #2D2D2D;
    border-width:1;
}

.graphElement.selected {
    border-color: #f0f0f0;
    border-width:1;
}

.node * {
    font:resource("Roboto/Roboto-Regular");
    font-size:12;
}

.node {
    border-width: 2;
    border-radius: 5;
}

.node > #pane {
    flex-direction: row;
    align-items: stretch;
    border-left: 4;
    border-top: 4;
```

VisualElement styles in a stylesheet

A `VisualElement` has style properties that can either be set directly in C# or applied from a stylesheet. In addition to layout properties, there also exist several properties affecting the way an element is drawn on screen, such as `backgroundColor` or `borderColor`.

Note: *in 2017.2, all style properties of* `VisualElement` *will be extracted to a separate data structure to remove ambiguity between style-driven properties and non-style properties.*

Style sheets are supported in Unity via the USS file format. Please refer to this document for detailed information about USS : Unity Style Sheets Reference.

Once you've successfully created your style sheets in Unity, you can associate a stylesheet to your VisualContainer via the `AddStyleSheetPath()` method.

Notes:

The style sheet needs to be placed inside an "Resources" or "Editor Default Resources" folder in your Assets folder

The path provided to the aforementioned method is relative to the enclosing folder

Style sheets modified while used in a running `EditorWindow` take effect immediately and should re-style your UI accordingly

Style sheets can be attached to any `VisualContainer` and thus their rules apply to all descendants of this container

Events

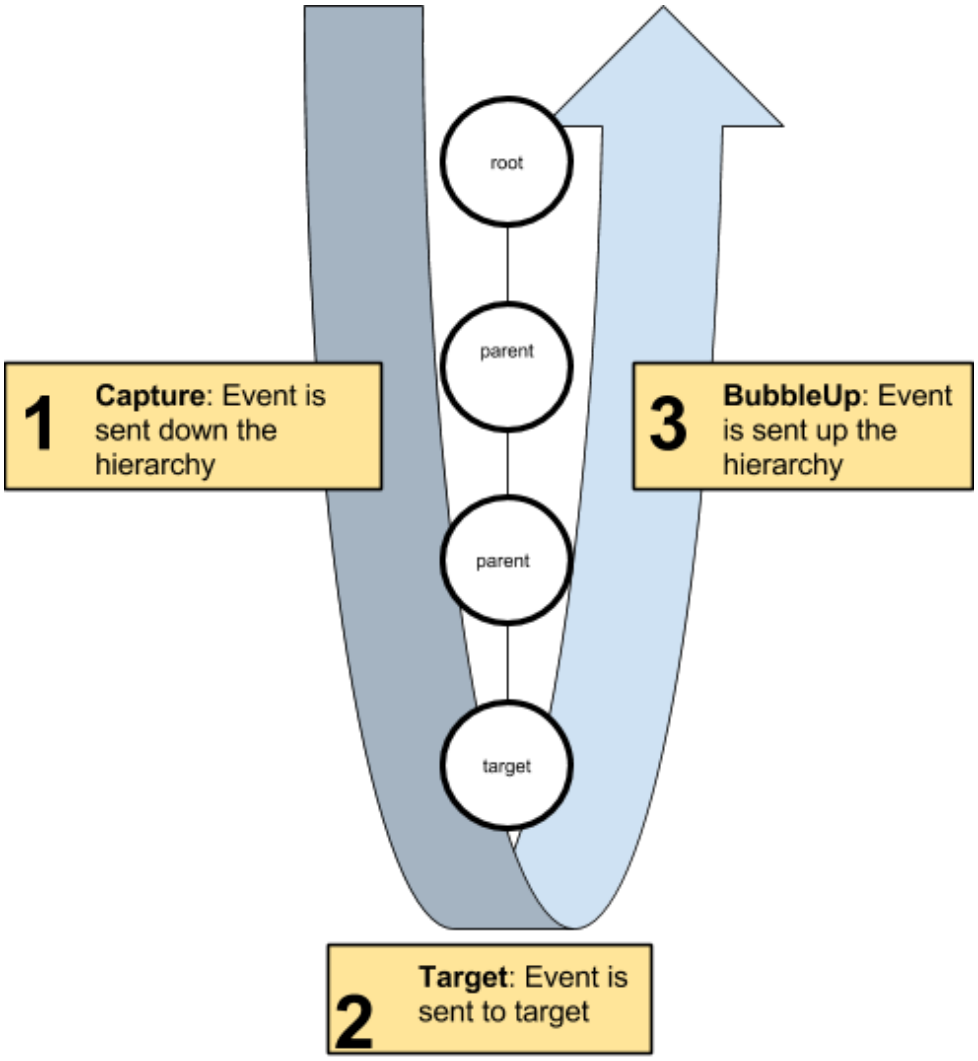
Input events are directly propagated to `VisualElement` objects. The logic varies slightly depending on the type of event and other factors :

Mouse events are propagated to the element under the mouse

Keyboard events are propagated to the focused element

A control can take the “capture” and receive all events directly (for example, a button takes the capture on mouse down and releases it on mouse up)

Events are said to be propagated because each ancestor of the target object can intercept an event before its descendants (Capture phase) or after its descendants if not stopped (BubbleUp phase). You might be familiar with this pattern from the HTML or Flash event models.



Event capture phase and bubble up phase

If an element has taken the capture (via the extension method `TakeCapture()`), it will receive the event directly and no propagation will occur. However, if it does not use the event it will be propagated normally.

An element that is not `visible` or `enabled` will not receive events. However, the events will still be propagated in the hierarchy..

Adding event handlers

There are two ways two write event handling code:

Override the `HandleEvent()` method in your `VisualElement` subclass

Subclass Manipulator and override the `HandleEvent()` method in your subclass, then add `VisualElement.AddManipulator()`

You must return `EventPropagation.Continue` or `EventPropagation.Stop` depending on how you want to control the propagation. The `finalTarget` argument will be set to the target object.

Note: *in 2017.2, we will move away from the IMGUI Event class and remove the need to write subclass to handle events. Instead, new event types will be introduced alongside an event registration model based on callbacks.*

Picking mode and custom shapes

The `VisualElement` class has a pickingMode property which supports two values :

`PickingMode.Position` (default) : perform picking based on the position rectangle

`PickingMode.Ignore` : prevent from being picked as the result of a mouse event

It's possible to override the `VisualElement.ContainsPoint()` method to perform custom intersection logic.

Controls

A few standard controls are built in UIElements :

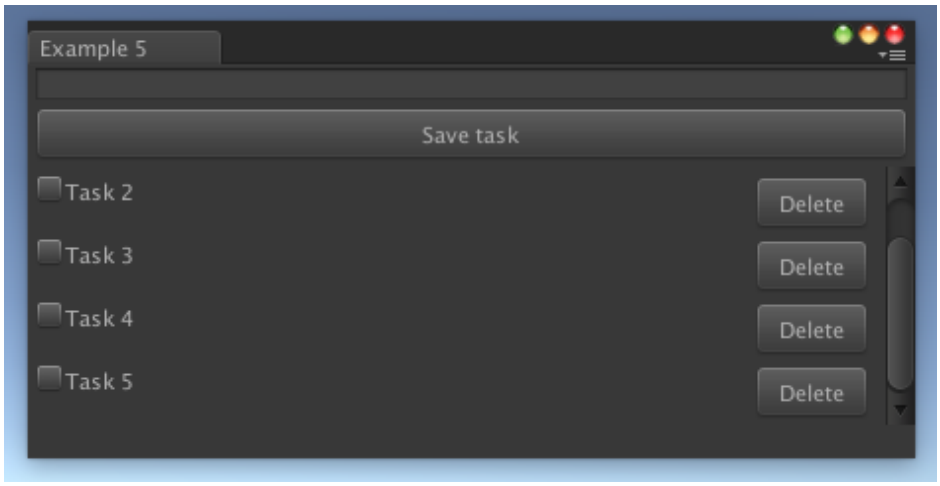
`Button`

`Toggle`

`Label`

`ScrollView`

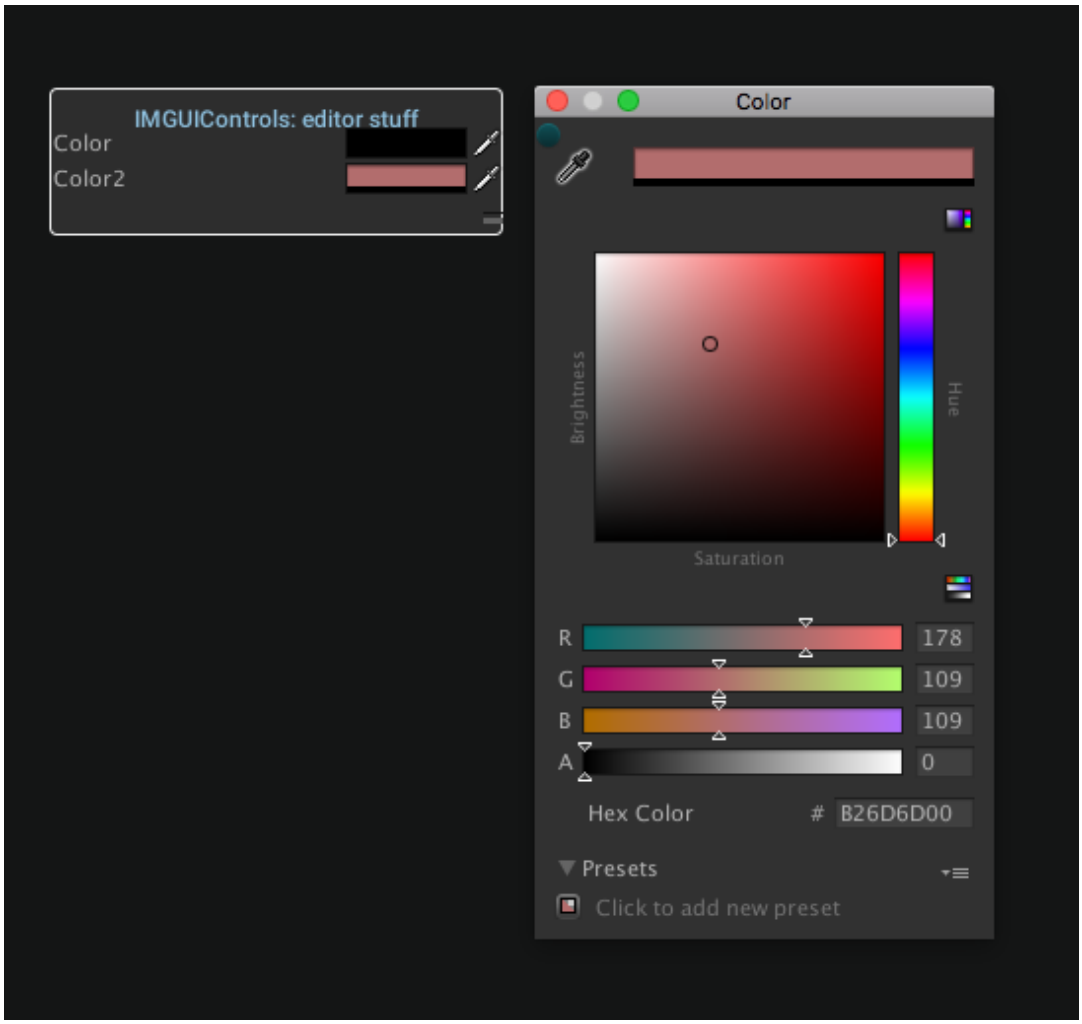
`TextField` and `EditorTextField`



Some of the standard controls built in to UIElements

Please refer to the example project for example usage of controls.

IMGUI Support



IMGUI code can be used inside of a VisualElement

The `IMGUIContainer` class lets you use IMGUI code inside of a `VisualElement`. Everything you can normally do inside of OnGUI() is supported. It's also possible to arrange multiple `IMGUIContainer` and lay them out correctly by mixing `GUILayout` and `UIElements` layout.

However, it's not possible use UIElements inside of an `IMGUIContainer`.

2017–08–08 Page published with no editorial review

New feature in Unity 2017.1

Scripted Importers

Leave feedback

Scripted Importers are part of the Unity Scripting API. Use Scripted Importers to write custom Asset importers in C# for file formats not natively supported by Unity.

Create a custom importer by specializing the abstract class ScriptedImporter and applying the ScriptedImporter attribute. This registers your custom importer to handle one or more file extensions. When a file matching the registered file extensions is detected by the Asset pipeline as being new or changed, Unity invokes the method `OnImportAsset` of your custom importer.

Note: Scripted Importers cannot handle a file extension that is already natively handled by Unity.

Note: Limitation

This is an experimental release of the Scripted Importer feature and as such is limited to assets that can be created using the Unity Scripting API. This is not a limitation of the implementation or design of the this feature, but does impose limits on its real-world use.

Example

Below is a simple example as Scripted Importer: It imports asset files with the extension “cube” into a Unity **Prefab** with a cube primitive as the main Asset and a default material with a color fetched from the Asset file:

```
using UnityEngine;
using UnityEditor.Experimental.AssetImporters;
using System.IO;

[ScriptedImporter(1, "cube")]
public class CubeImporter : ScriptedImporter
{
    public float m_Scale = 1;

    public override void OnImportAsset(AssetImportContext ctx)
    {
        var cube = GameObject.CreatePrimitive(PrimitiveType.Cube);
        var position = JsonUtility.FromJson<Vector3>(File.ReadAllText(ctx.assetP

        cube.transform.position = position;
        cube.transform.localScale = new Vector3(m_Scale, m_Scale, m_Scale);

        // 'cube' is a a GameObject and will be automatically converted into a p
        // (Only the 'Main Asset' is elligible to become a Prefab.)
        ctx.AddObjectToAsset("main obj", cube);
        ctx.SetMainObject(cube);

        var material = new Material(Shader.Find("Standard"));
        material.color = Color.red;
```

```
        // Assets must be assigned a unique identifier string consistent across
        ctx.AddObjectToAsset("my Material", material);

        // Assets that are not passed into the context as import outputs must be
        var tempMesh = new Mesh();
        DestroyImmediate(tempMesh);
    }
}
```

Note:

The importer is registered with Unity's Asset pipeline by placing the the `ScriptedImporter` attribute on the CubeImporter class.
The CubeImporter class implements the abstract `ScriptedImporter` base class.
`OnImportAsset`'s ctx argument contains both input and output data for the import event.
Each import event must generate one (and only one) call to `SetMainAsset`.
Each import event may generate as many calls to `AddSubAsset` as necessary.
Please refer to the Scripting API documentation for more details.
You may also implement a custom Import Settings Editor by specializing ScriptedImporterEditor class and decorating it with the class attribute `CustomEditor` to tell it what type of importer it is used for.

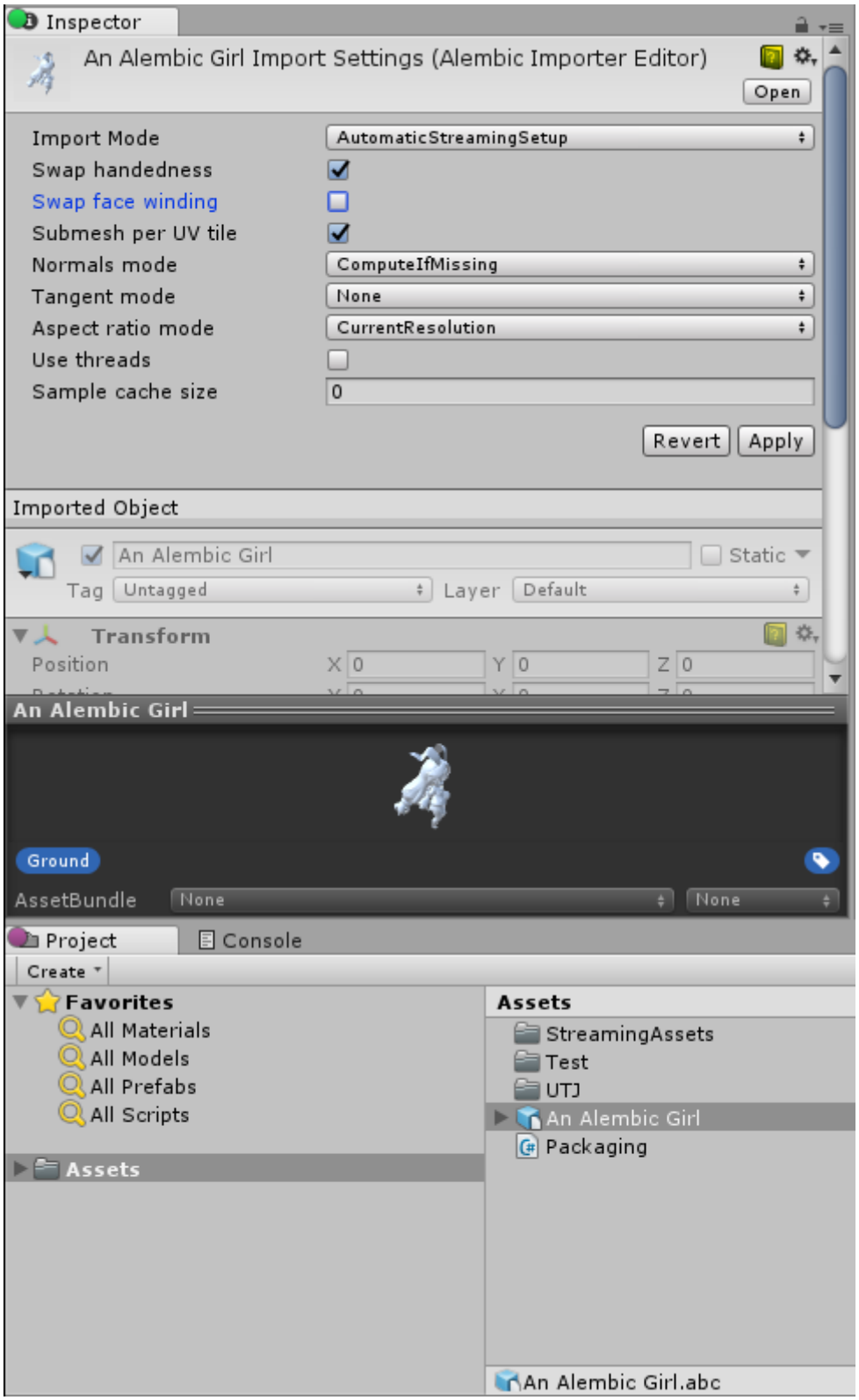
For example:

```
using UnityEditor;
using UnityEditor.Experimental.AssetImporters;
using UnityEditor.SceneManagement;
using UnityEngine;

[CustomEditor(typeof(CubeImporter))]
public class CubeImporterEditor: ScriptedImporterEditor
{
    public override void OnInspectorGUI()
    {
        var colorShift = new GUIContent("Color Shift");
        var prop = serializedObject.FindProperty("m_ColorShift");
        EditorGUILayout.PropertyField(prop, colorShift);
        base.ApplyRevertGUI();
    }
}
```

Using a Scripted Importer

Once you have added a scripted importer class to a project, you may use it just like any other native file type supported by Unity:

Drop a supported file in the Asset directory hierarchy to import.
Restarting the Unity Editor reimports any files that have changed since last update.
Editing the Asset file on disk and returning to the Unity Editor triggers a reimport.
Import a new asset using **Asset** > **Import New Asset...**.
Explicitly trigger a re-import via the menu: **Asset** > **Reimport**.
Click on the Asset to see its settings in the Inspector window. To modify its settings, edit them in the **Inspector** window and click **Apply** .



The Inspector window of an Asset (An Alembic Girl) imported by the Scripted Importer

Real-world use of Scripted Importers

Alembic: The Alembic importer plug-in has been updated to use a Scripted Importer. For more information, visit Unity github: AlembicImporter.

USD: The USD importer plug-in has been updated to use a Scripted Importer. For more information, please visit Unity github:: USDForUnity.

New feature in 2017.1

2017–07–27 Page amended with no editorial review

Look Dev

Leave feedback

Introduction

Look Dev is an experimental image-based lighting tool that allows you to check and compare different **Assets** through a viewer to ensure they are correctly authored for various lighting conditions. It also allows you to debug **Assets** where required. Load **Assets** in as **Prefabs** for the best experience.

Asset validation

Asset validation is the process of confirming that Assets are authored correctly so they behave as expected in different lighting environments.

You must validate using **HDRI**. **HDRI** is a type of real-world lighting with incredibly high detail. As such, it offers a perfect lighting rig that would be difficult to create by hand. By using such an accurate lighting environment to test an Asset, you can determine whether the Asset itself or the game's lighting is causing any poor visual quality.

To learn more about creating HDRI environments, see documentation on the HDRI view.

Look Dev allows you to load two different Assets in two different views. This is useful for comparison. For example, an Art Director can check that a new Asset matches the art direction guidelines of a reference Asset.

Who should use Look Dev?

Look Dev is mainly designed for texture artists, modelling artists, lighting artists, artistic directors, outsourcing managers and anybody else involved in the visual art style of a project.

Rather than using multiple tools to visualise and validate new content, Look Dev allows everybody to visualize the Asset the same way. This means that you can validate the Asset immediately because how it appears in the tool is the same as how it appears in the finished project.

Note that while Look Dev is intended for artists, it is not an artistic tool. It is only intended to be used for Asset visualization and validation.

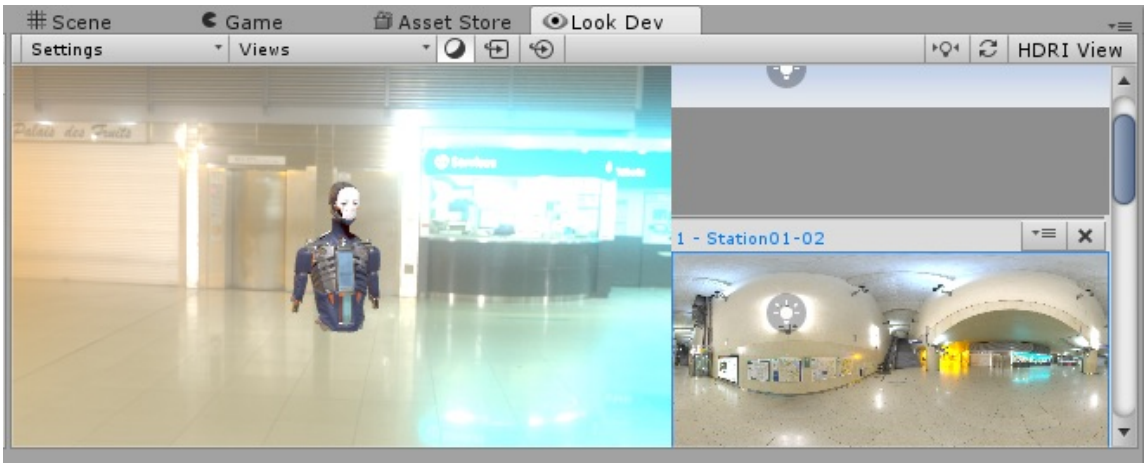
Initial setup

Before using Look Dev, you need to make sure Unity is set up for the best results:

Go to **Player Settings** (menu: **Edit** > **Project Settings** > **Player**) and navigate to **Other Settings**.
Set the **Rendering Path** to **Deferred** and the **Color Space** to **Linear**.
Look Dev always uses these settings, independently of the project settings, so changing these settings in your project ensures that what you see in the Look Dev viewer matches what you see in the **Scene view**.

Known issues

When a **skybox** uses a cube map with a specular convolution type, it produces a blurry result in smaller **rendering** windows. Use a large window when using Look Dev in this context.

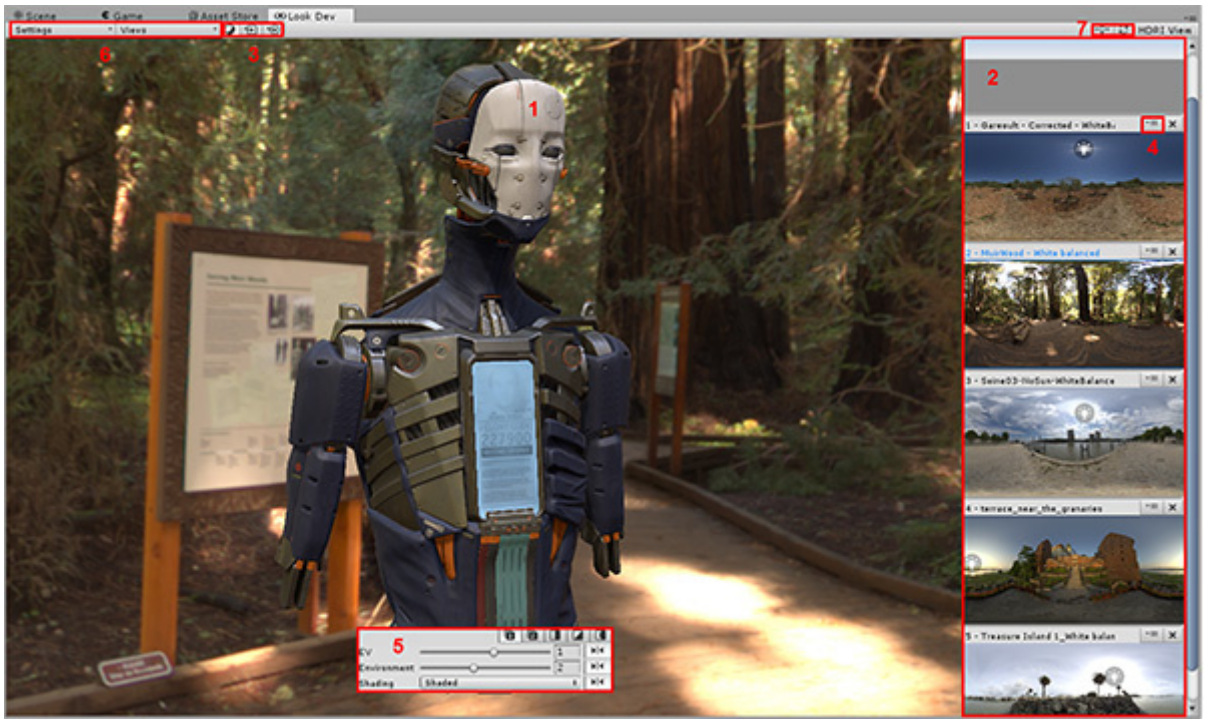


Known issue: skybox can produce a blurry result in certain circumstances

Overview

Leave feedback

To access Look Dev in Unity, go to **Window** > **Experimental** > **Look Dev**.



Look Dev view

HDRI view

Shortcut buttons for Environment shadow and Look Dev view

HDRI menu

Control panel

Settings and Views menus

Shortcut buttons for HDRI view

Controlling the camera

The Look Dev **camera** largely shares its behavior with the **camera** in the **Scene view**:

Rotate around pivot: Left click and drag (this is the same as in Scene view, except Look Dev doesn't require Alt+left click)

Pan camera: Middle click and drag

Zoom: Alt+right click and drag

Forward/backward: Mouse wheel

FPS mode: Right click + WASD

Keyboard shortcuts

In Look Dev view:

Rotate environment: Ctrl+left click and drag

Rotate directional light for environmental shadow: Shift+left click and drag

Swap environment: Left/right arrow keys

Manipulate the separator: Hold Shift to rotate it in increments

Frame camera to the observed object: F

In **HDRI** view:

Rotate environment: Ctrl+left click and drag

The Look Dev view

Leave feedback

The Look Dev view is the main window. You can open multiple Look Dev views at once.

By default there is only one view in the Look Dev view, but you can choose from a selection of split-screen views. See Multi-view below to learn more about this.

Loading Prefabs into the Look Dev view

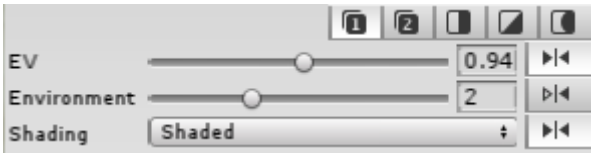
The Look Dev view is primarily designed to view **Asset Prefabs**.

To load a Prefab into the Look Dev view, right click the Prefab and select **Open in Look Dev Tool**. Alternatively, if Look Dev is already open, drag and drop the Prefab from the Hierarchy window directly into the Look Dev view.

Note: When you right click on a Prefab to load it into Look Dev, it automatically loads into the currently visible Look Dev view or the last active one.

Multi-view

By default, the Look Dev view displays a single pane containing the Prefab you are working with. Use the buttons in the Control Panel to display a split-screen duplicate pane alongside it, allowing you to compare and contrast different settings.

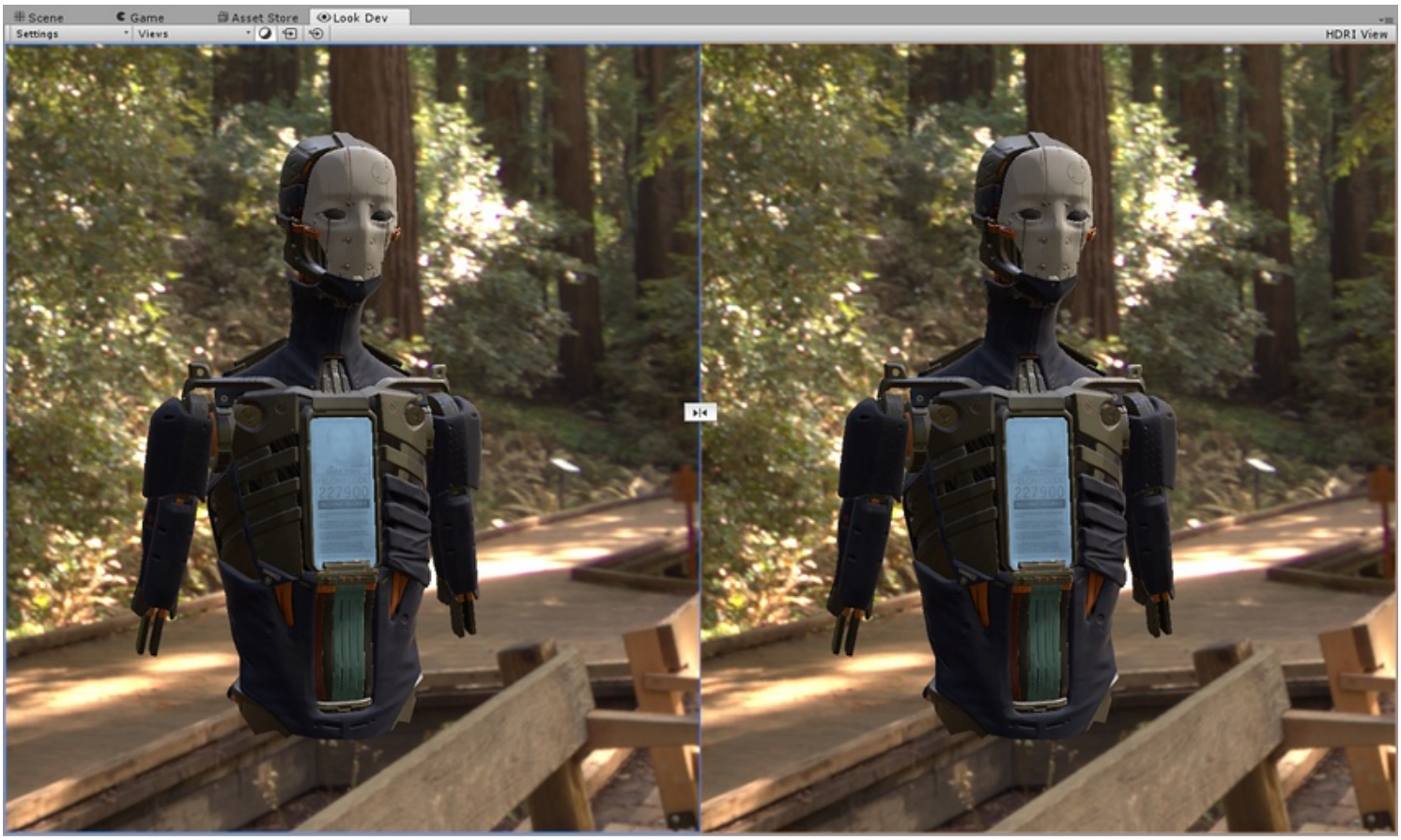


Control panel

When you select the button containing the number **1** or number **2** , the Look Dev view is in single-pane mode. Each button corresponds to its view in Look Dev. Select button **1** to apply settings to view 1, and button **2** to apply settings to view 2. The other three buttons are as follows:

Side-by-side:

Duplicate views displayed side by side.

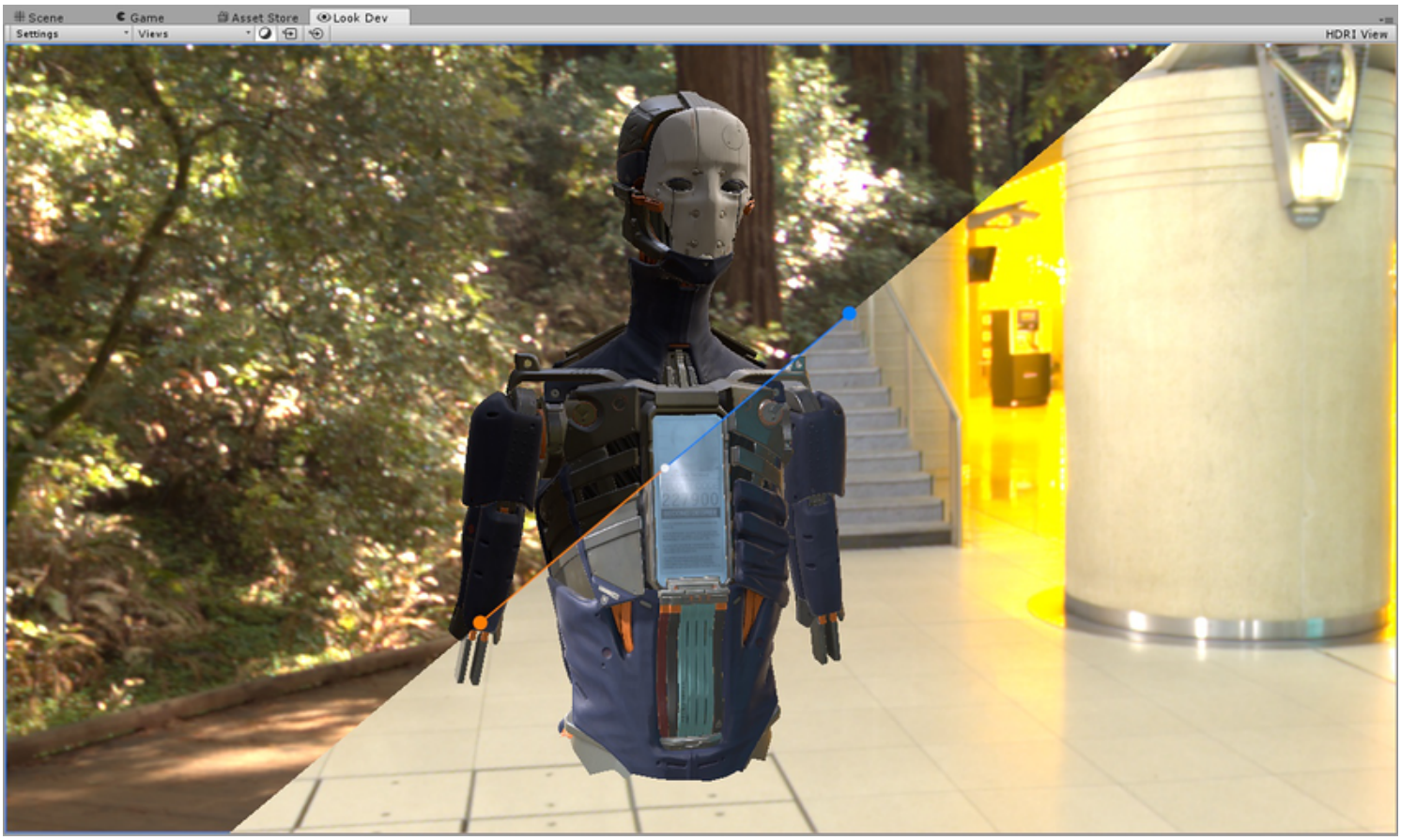


Side-by-side view

By default, both views use the same **camera**. The lock button at the center decouples this behavior, allowing you to manipulate the camera in both views independently.

Split-screen:

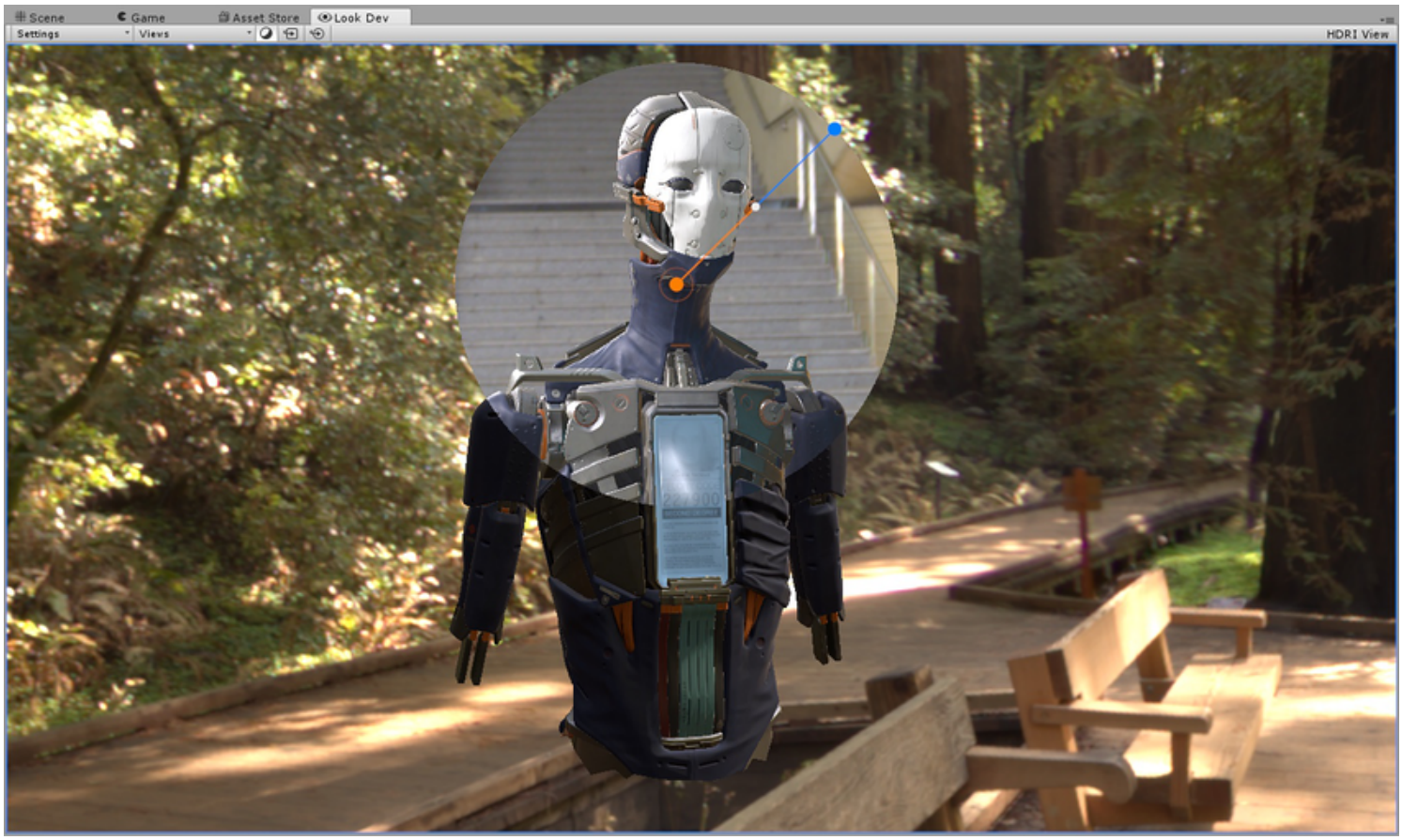
The view is split horizontally in two, and an orange/blue manipulation **Gizmo** represents the separation plane between the two views (see Using the manipulation Gizmo below for guidance on using this).



Split-screen view

Zone

The view is split in two, with a circular split defined by the position of the orange/blue manipulation Gizmo:



Zone view

To load a different Prefab into each split-screen view, go to **Settings** and enable **Allow Different Objects**, then drag and drop each Prefab into its intended view.

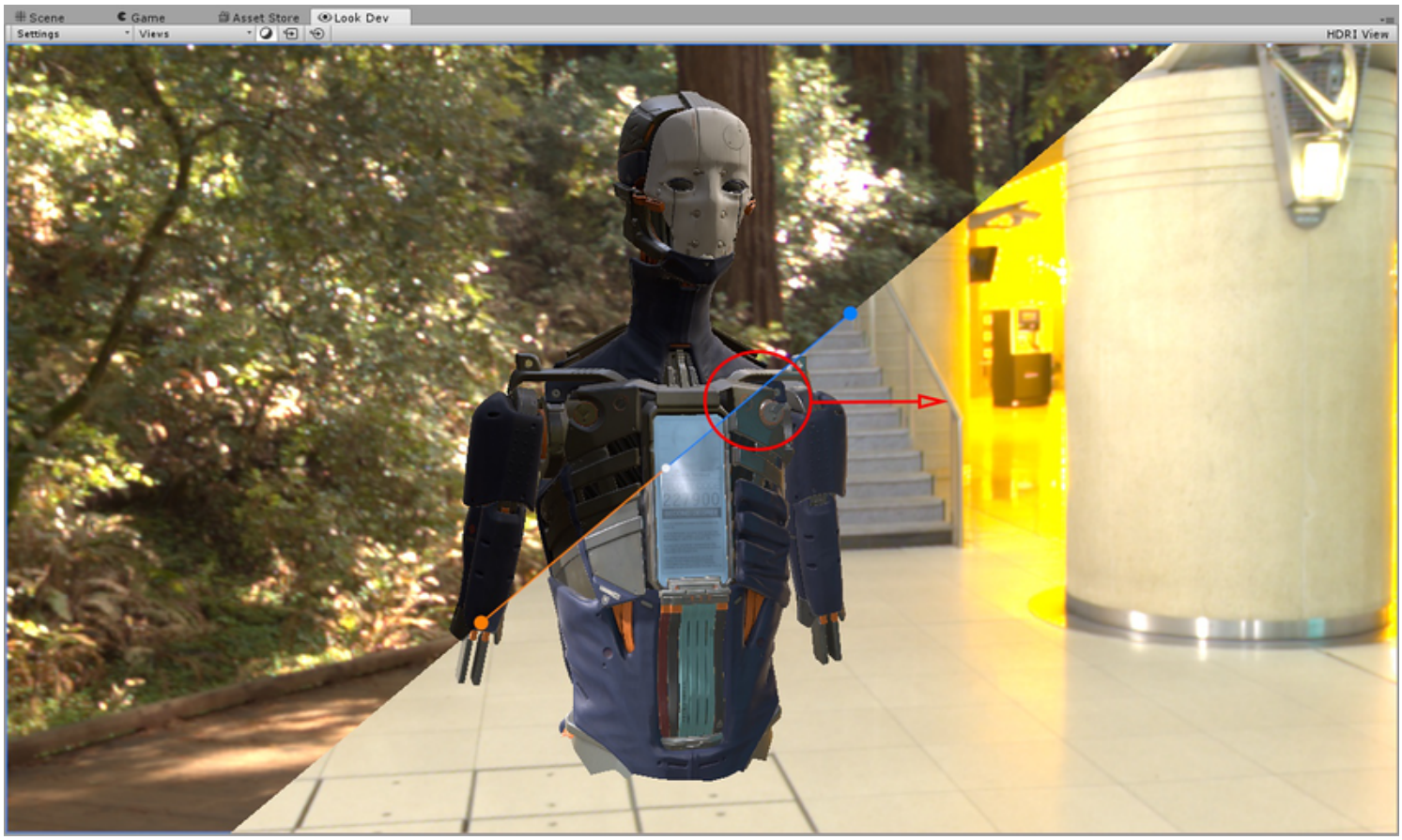
Note that in multi-view modes, it only makes sense to do this if you need to look at two different versions of the same Asset. Comparing completely different Assets doesn't give you a good idea of the difference in lighting or visual effect.

Using the manipulation Gizmo

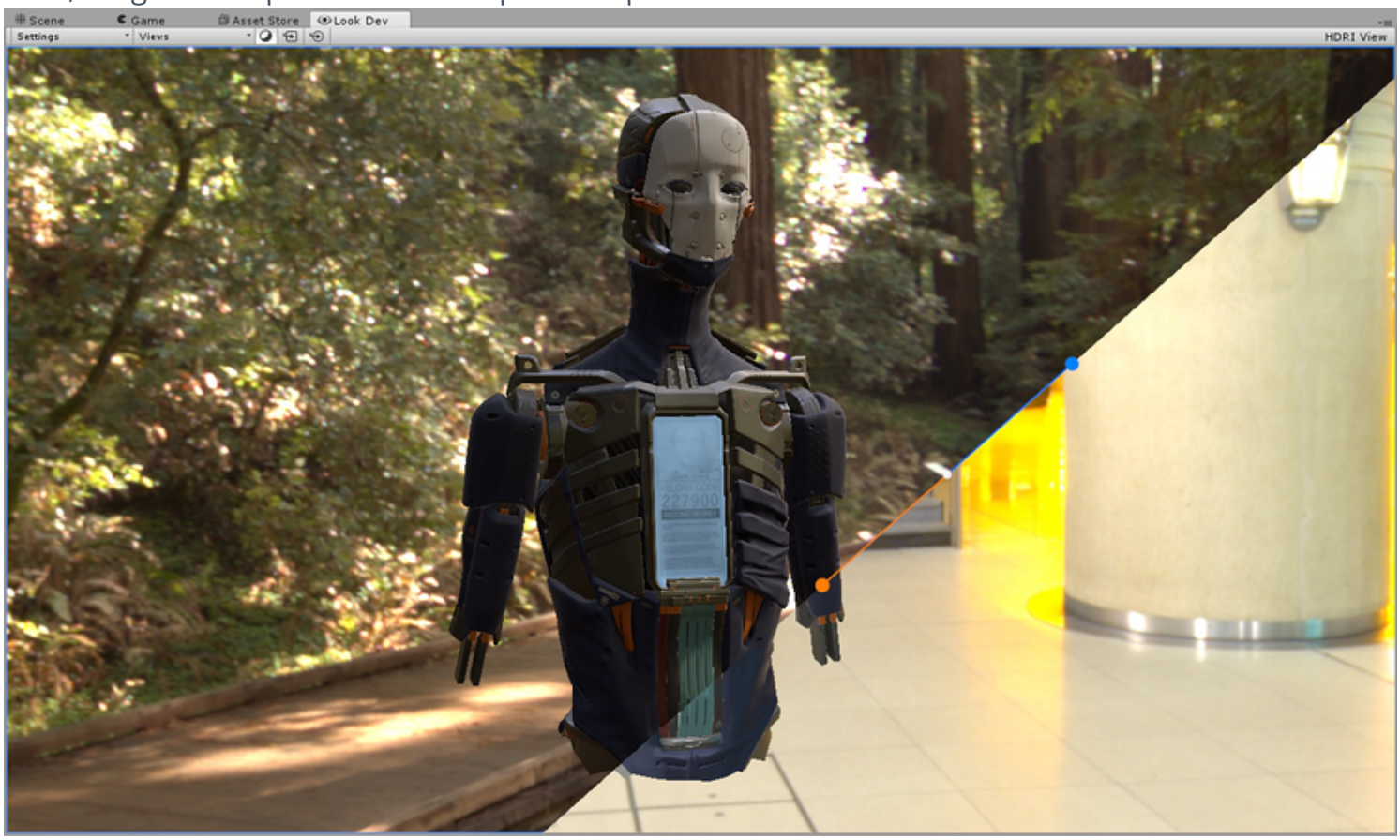
The manipulation Gizmo has different behaviour in Split-screen and Zone modes, but you use it in the same way in both.

Moving the separator

To move the separator, click on the narrow part of the Gizmo and drag and drop it to the location you want.



Click, drag and drop to move the separation pane

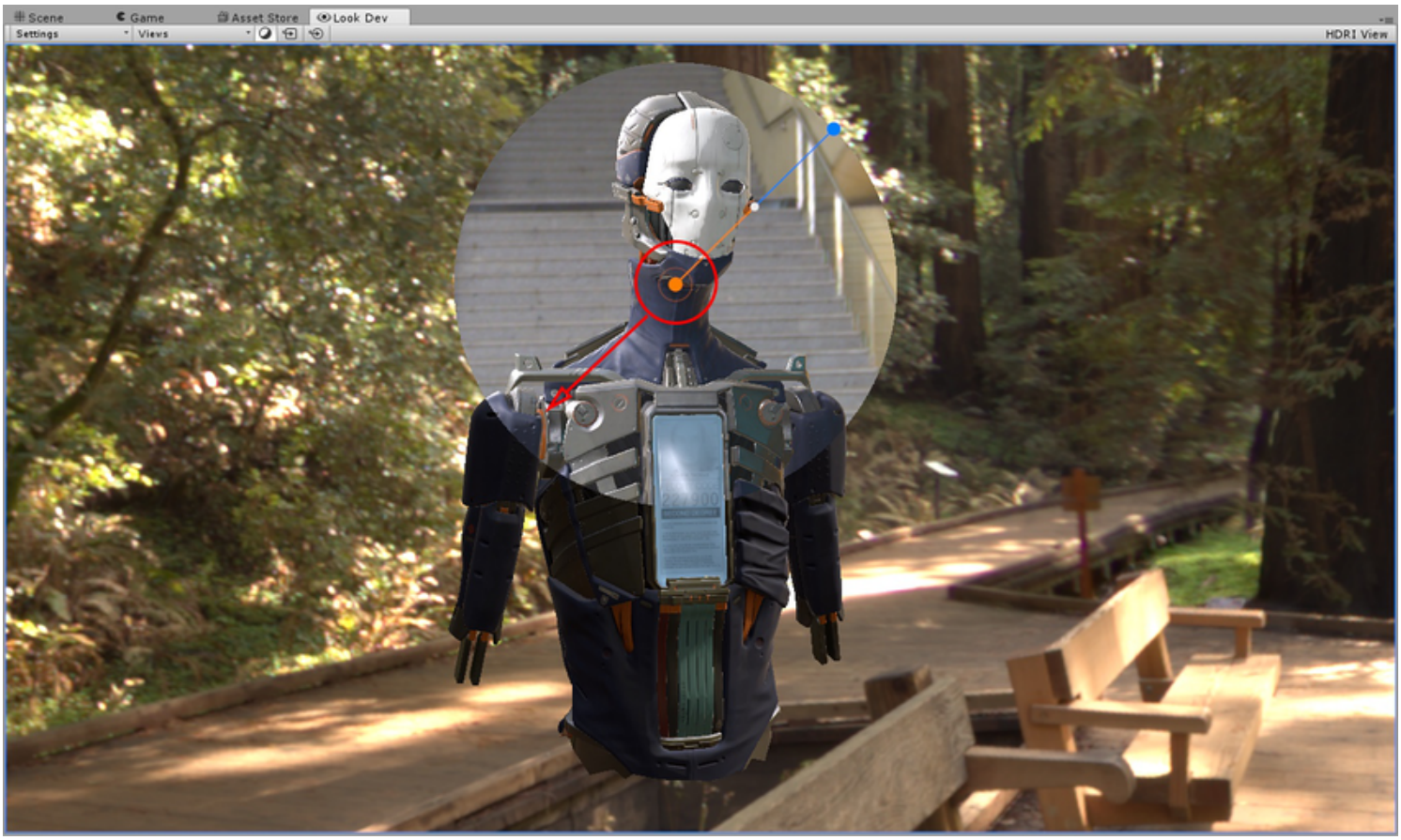


The separation plane dragged to the right

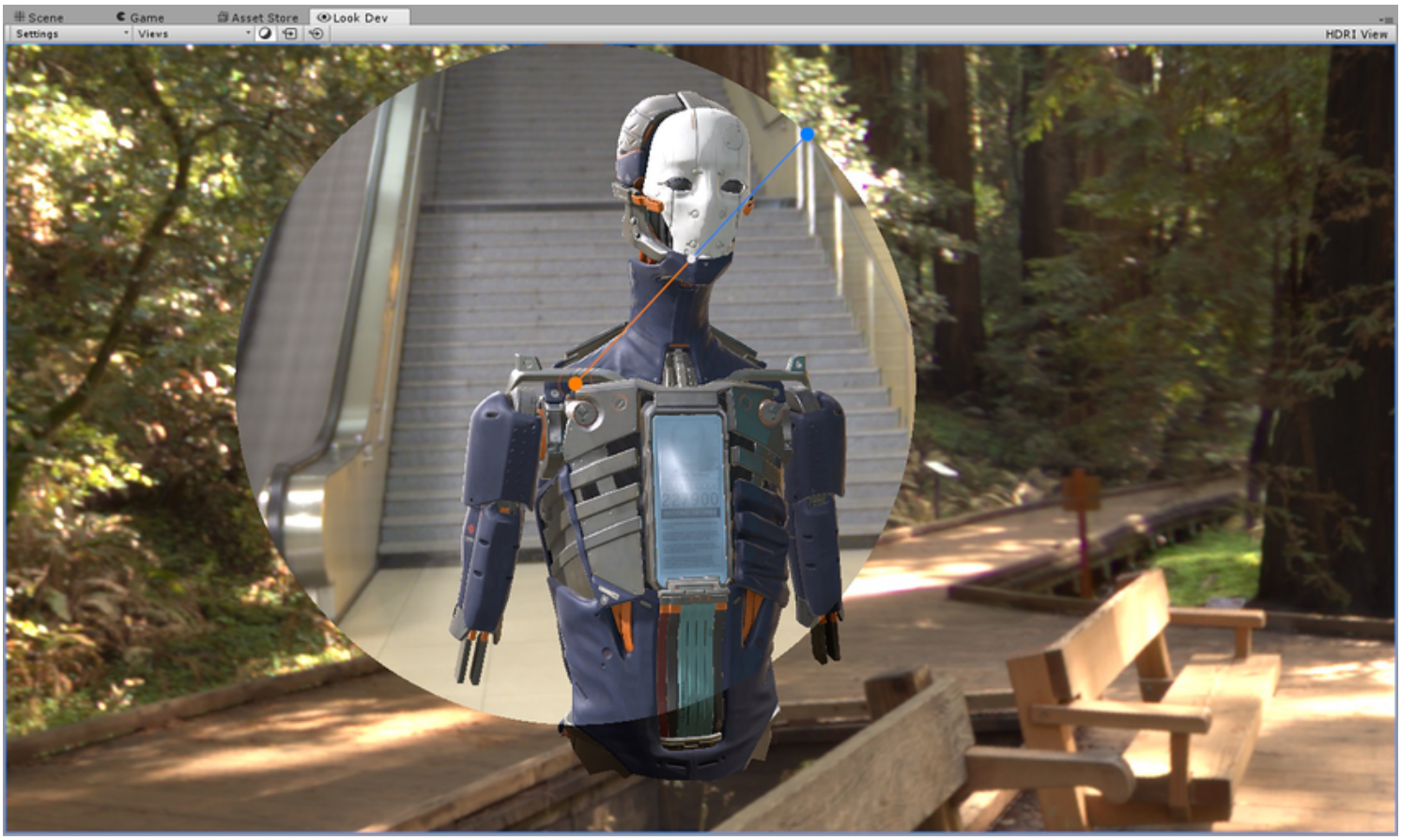
Changing the orientation and length

To change the orientation and length of the manipulator Gizmo, click and drag the circle at either end of the manipulator.

In Zone mode, moving the circles on the manipulator Gizmo changes the radius of the view circle.



How to move the circles on the manipulator Gizmo



Moving the Gizmo circles has made a larger radius of the view circle

Changing the separation in increments

To change the separation in increments, click and hold the circle on the end of the manipulation Gizmo, then hold **Shift** as you move the mouse. This snaps the manipulation Gizmo to set angles, which is useful for a perfectly horizontal, vertical or diagonal angle.

The central white circle on the separator allows you to blend between the two views. Left click on it and drag it along the orange line to blend the left-hand view with the right-hand view (as shown in Image A below). Drag to the blue line to blend the right-hand view with the left-hand view (as shown in Image B below).

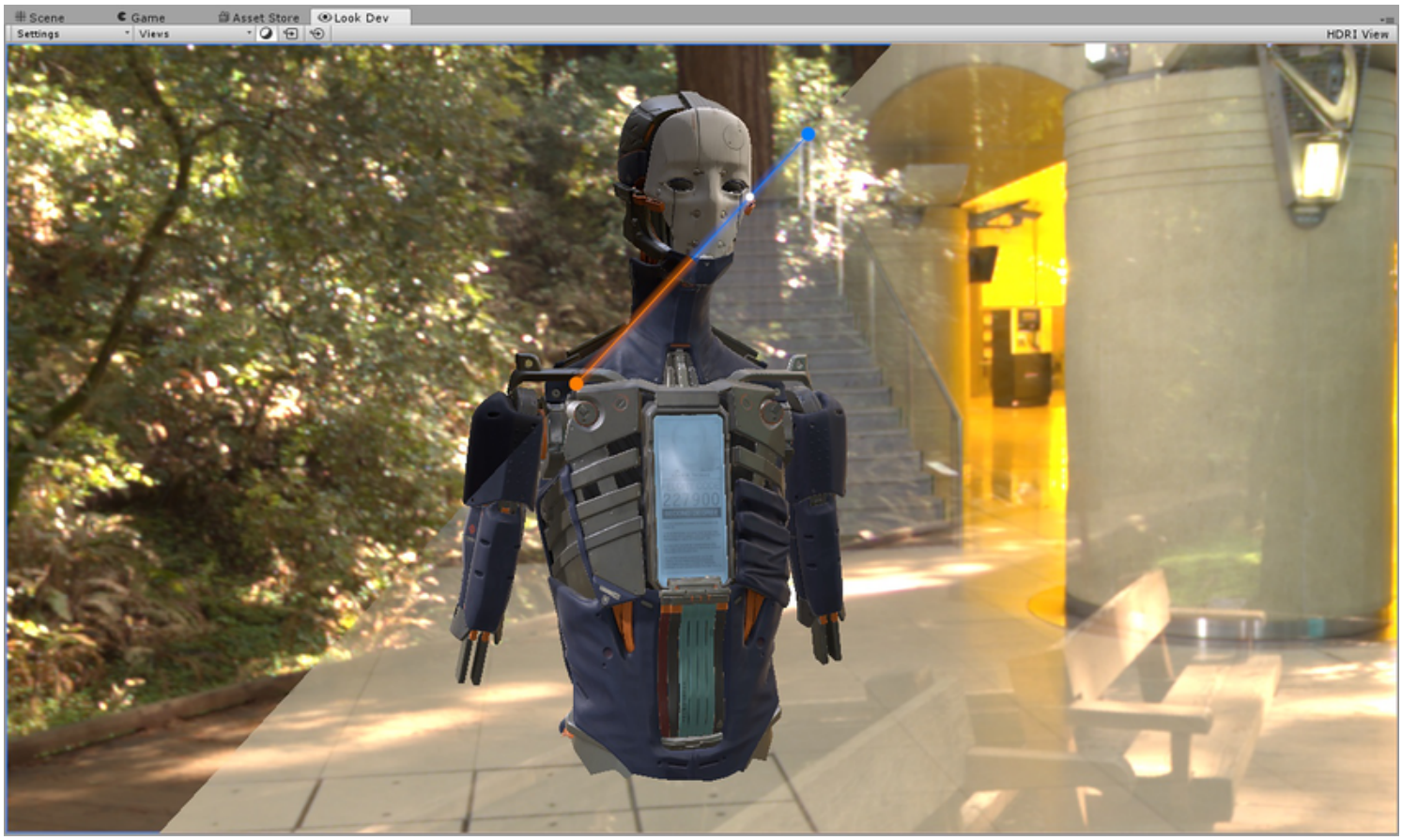


Image A

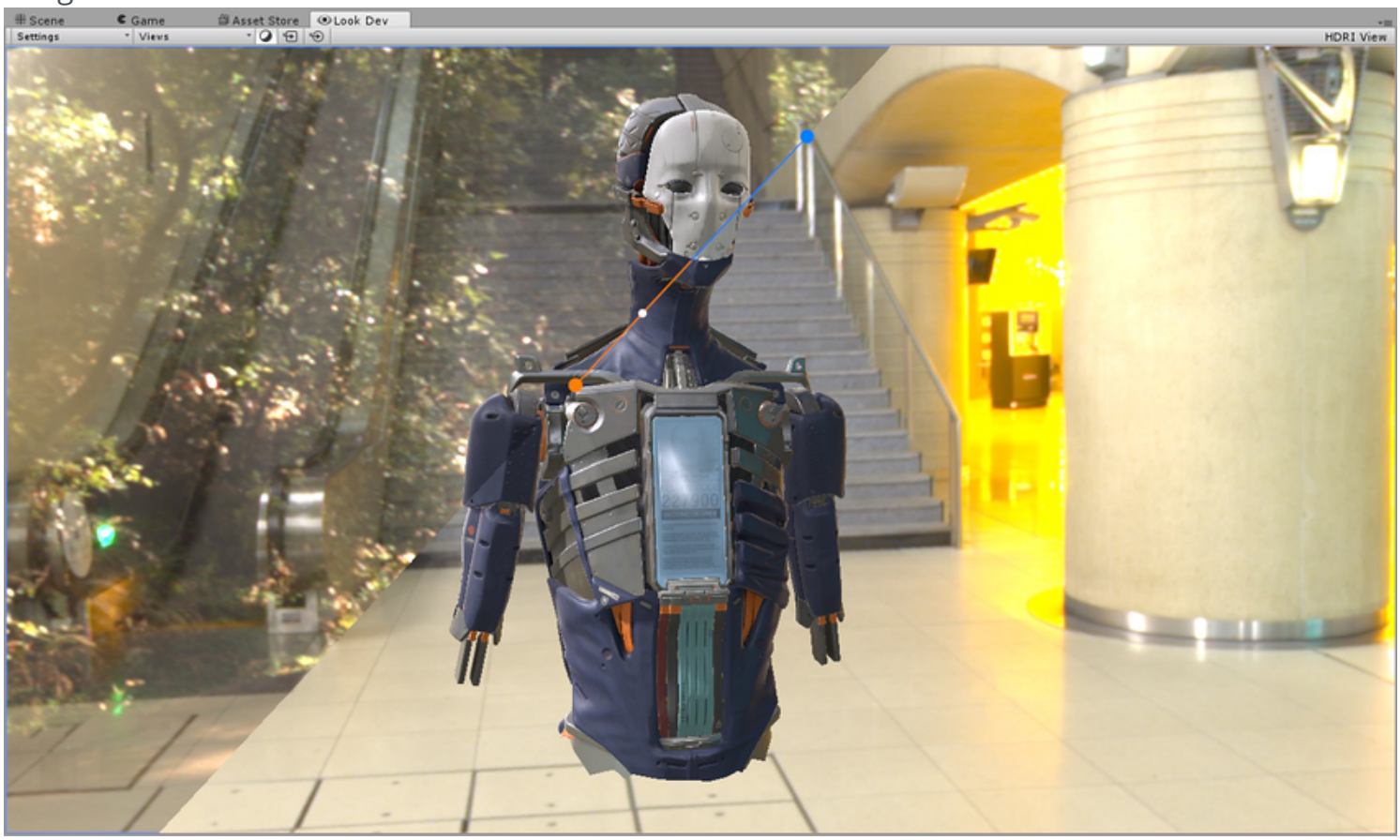


Image B

The white circle automatically snaps back into the center when you drag it back. This helps you get back to the default blending value quickly.

Uses for each option

Side-by-side: Compare two different lighting conditions on the same Asset to check that it behaves correctly.

Split-screen/Zone: Investigate texture problems using a debug **Shader** mode (for example, use one screen to view Normal or Albedo shading, and the environment-lit mode on the other).

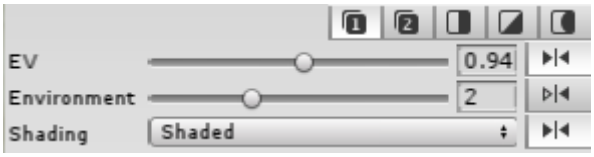
Side-by-side/Split-screen/Zone: Compare two different versions of the same Asset using the same lighting conditions to see which changes improve the Asset's quality.

Side-by-side/Split-screen/Zone: Compare two different Asset LODs using the same lighting conditions, to test the quality of the **LOD**.

Control panel

Leave feedback

The **Control Panel** gathers all the most commonly used settings in the Look Dev Views menu, for easy and fast access.



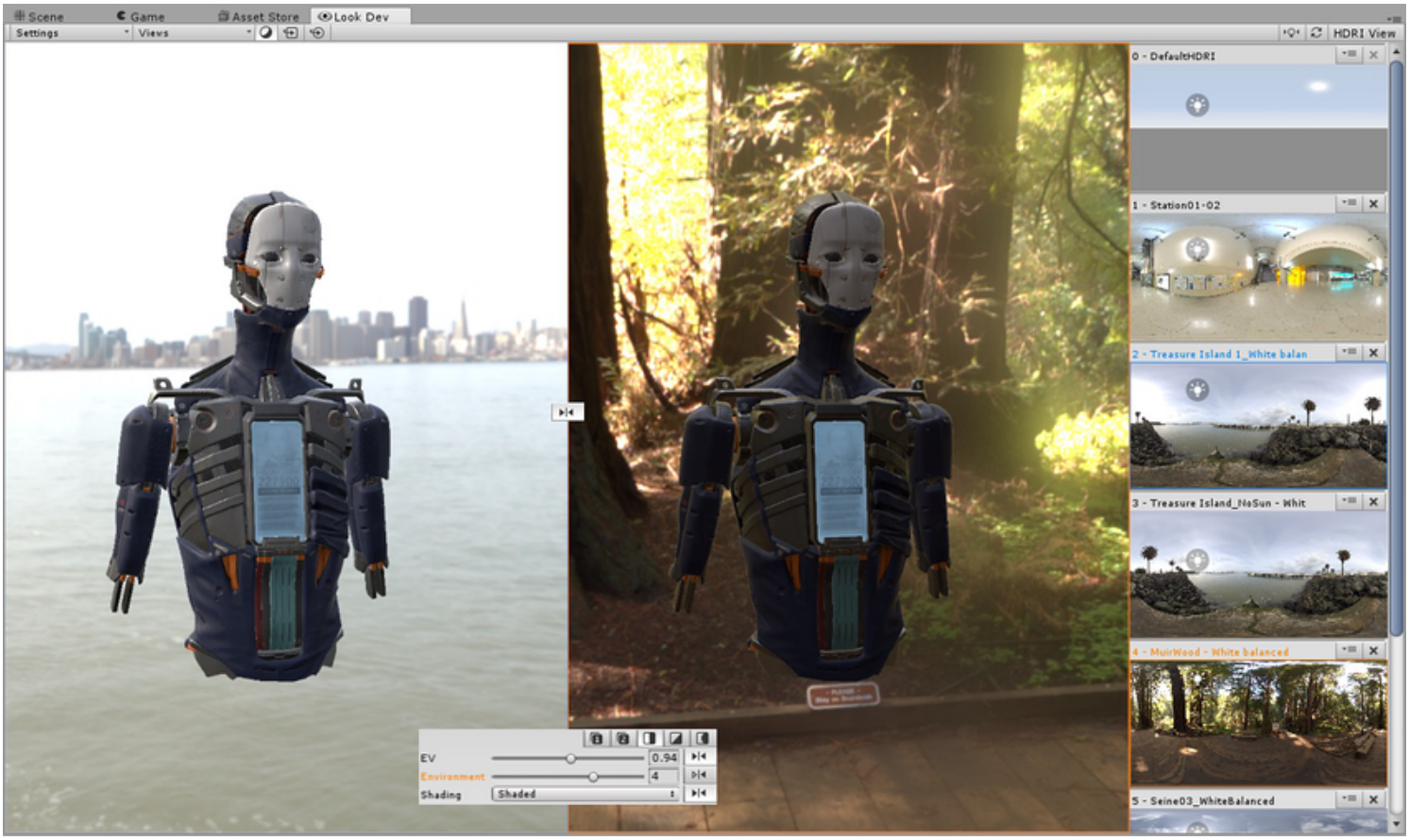
Control Panel

Split-screen and Zone modes

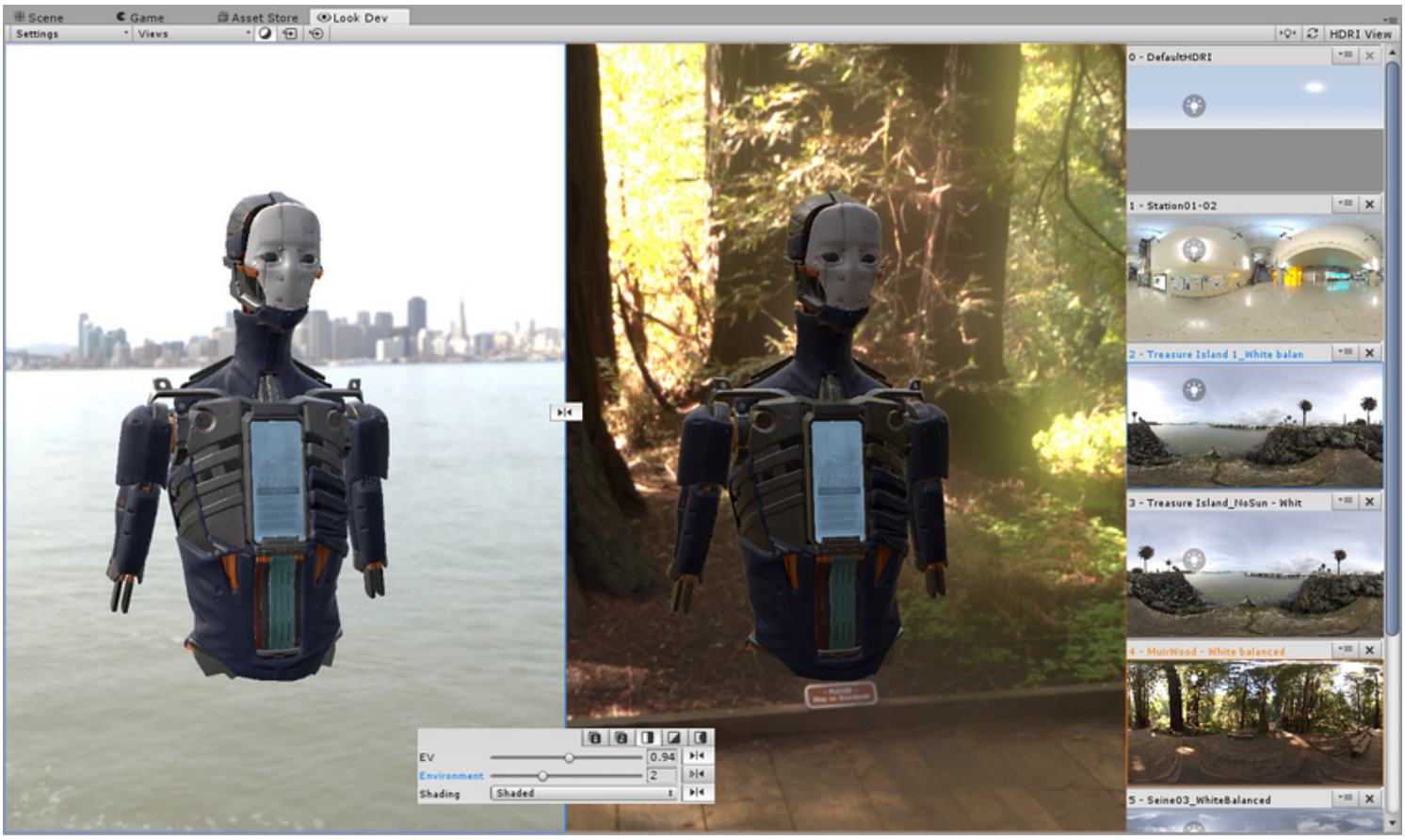
In **Split-screen** and **Zone** mode, all properties are synchronized between the two views by default. Use the **lock** button (highlighted in the image below) to make a property independent and applied only to the selected view (indicated by the blue or orange border around the border of the view).

Control Panel with the **lock** button highlighted

Properties in the **Control Panel** that only apply to one window are highlighted in the color of the selected view for further clarity.



Here, the **Environment** setting applies to the orange highlighted window only

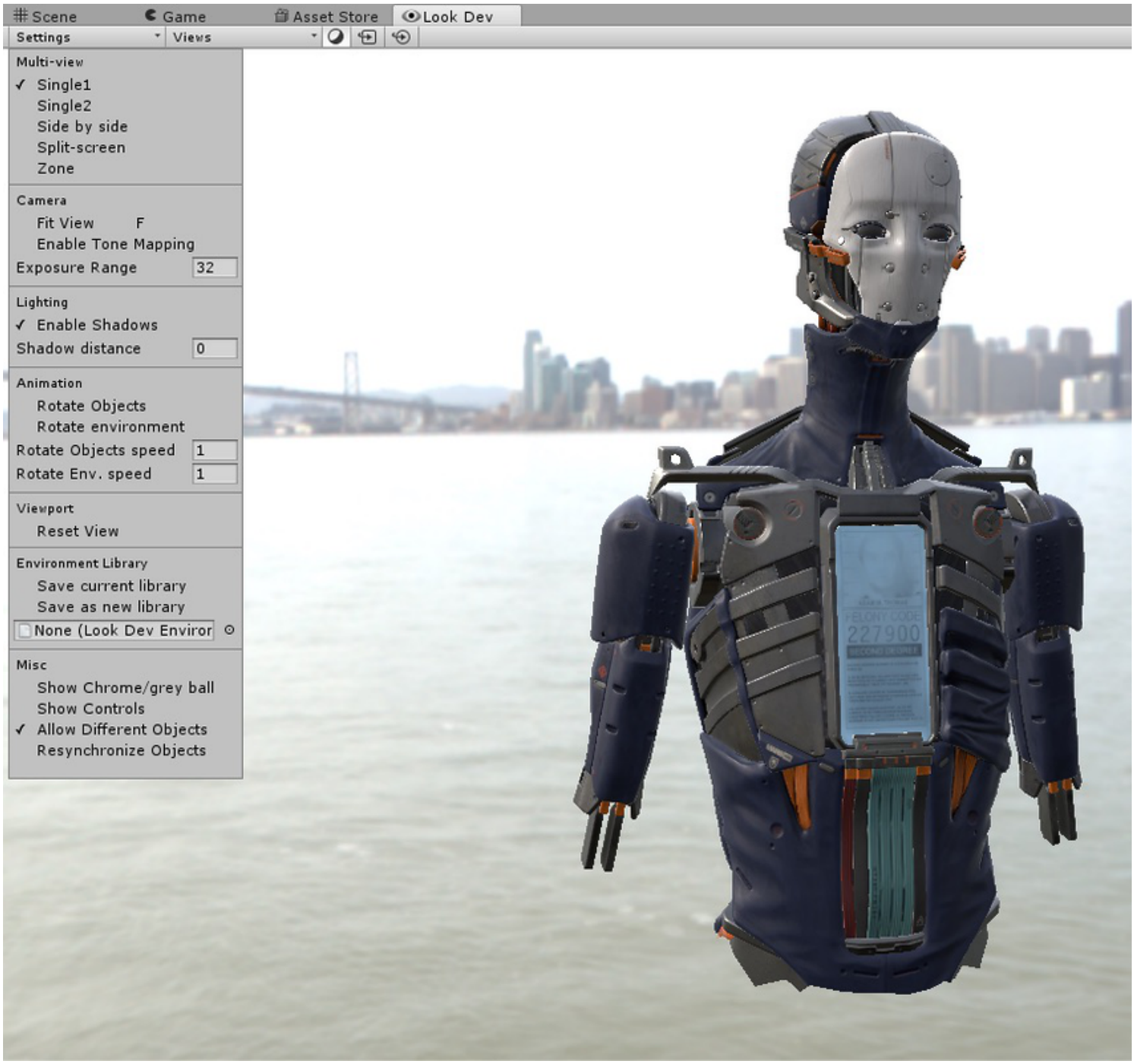


Here, the **Environment** setting applies to the blue highlighted window only

Settings menu

Leave feedback

The **Settings** menu is located in the top left of the Look Dev view. This contains all of the global options for Look Dev.



Multi-view

Property	Function
Single	A single-screen view of Look Dev.
Side by side	Duplicated view, displayed side by side.
Split-screen	Splits the view horizontally in two. An orange/blue manipulation **Gizmo** represents the separation plane between the two views.

Property	Function
Zone	Splits the view in two, with a circular split defined by the positioning of the orange/blue manipulation Gizmo.

Camera

Property	Function
Fit View	Click this to reset the position of the objects to the center of the view (or views, if there are multiple views).
Enable Tone Mapping	Click this to toggle the built-in tone mapper. This tone mapper is not an artistic tone mapper but a neutral one which conserves the range but doesn't include any film effects.
Exposure Range	Use this to define the minimum and maximum range of the **Exposure Value** (EV) slider in the Control Panel and Views menu (up to a maximum of 32). Useful for testing an extreme light source in the **HDRI**.

Lighting

Property	Function
Enable Shadows	Click this to toggle Environment Shadow off and on.
Shadow distance	Use this to set the maximum shadow distance for the Environment Shadow. If this is set to 0, an automatic number is defined based on **Mesh** bounds.

Animation

Property	Function
Rotate Objects	Enable this to make an **Asset** or **GameObject** continuously rotate in the Look Dev view.
Rotate environment	Enable this to make the HDRI environment continuously rotate in the Look Dev view.
Rotate Objects speed	Use this to set the speed of the GameObject's rotation (if **Rotate Objects** is enabled).
Rotate Env. speed	Use this to set the speed of the environment's rotation (if **Rotate environment** is enabled).

Viewport

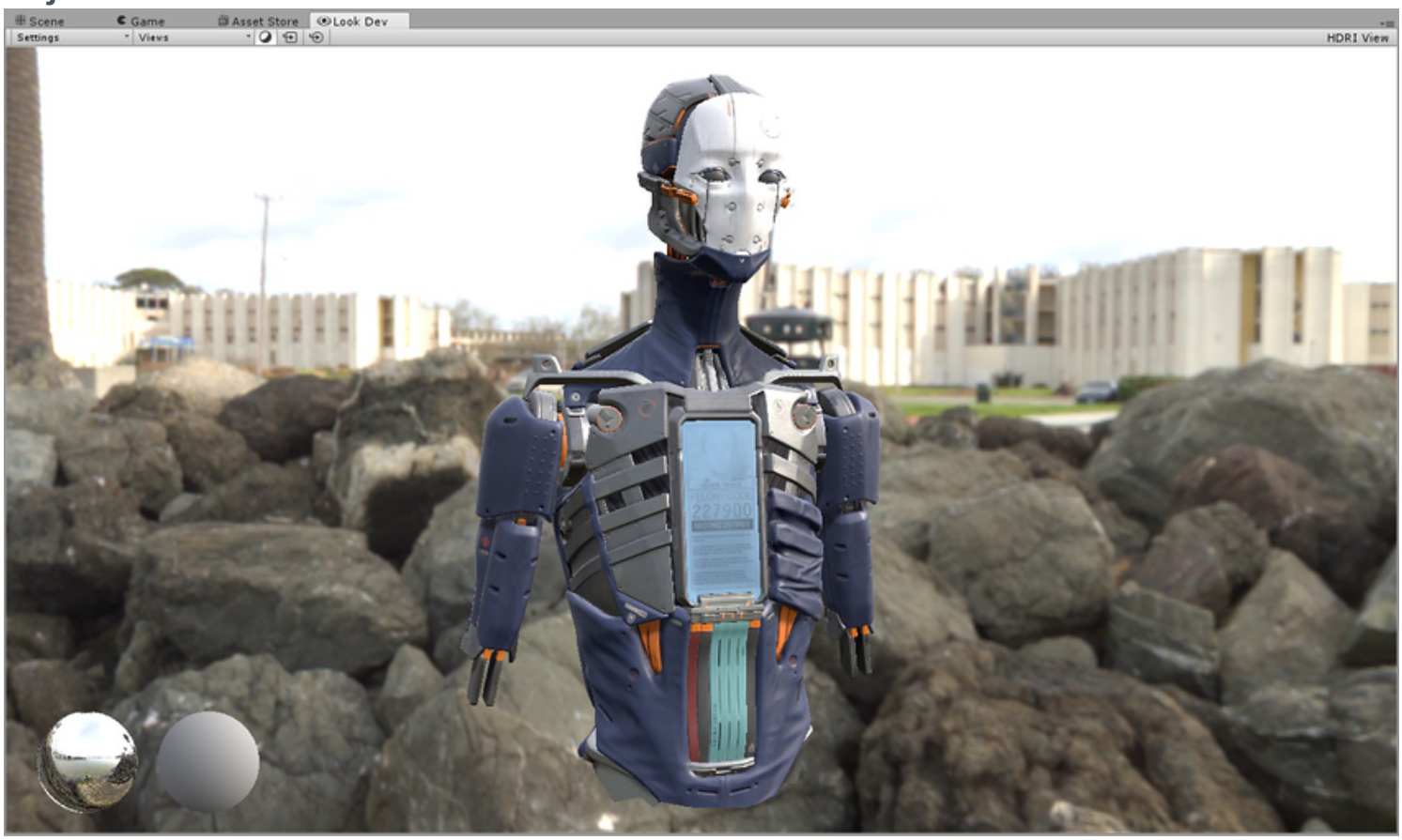
Property	Function
Reset View	Click to reset the **Camera** settings and **View** settings.

HDRI Library

Property	Function
Save Current Library	Select this to save the current HDRI Library.
Save As New Library	Select this to save the current HDRI Library to a new Asset Library.
Load Library	Load a previously saved HDRI Library. The field here displays the name of the current HDRI Library Asset that has been loaded.

Misc

Property	Function
Show Chrome/grey ball	Enable this to display a chrome ball and a grey ball in the bottom left of the screen (see image below). This is useful for debugging reflections and checking the average lighting reference.
Show Controls	Enable this to display the Control Panel in the Look Dev view.
Allow Different Objects	Enable this to load two different GameObjects when in multi-view mode.
Resynchronise Objects	Use this to assign the current GameObject in your selected view to the other view.

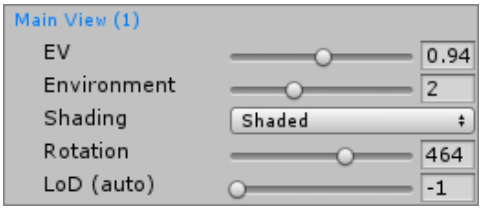


The user interface with **Show Chrome/grey ball** enabled

Views menu

Leave feedback

The **Views** menu is located in the top-left corner of the Look Dev view, next to the **Settings** menu. This controls all the options specific to each individual view.



Views menu

Property	Function
Shading	Select the drop-down menu to choose the shading mode. Shaded is the default. Unity-supported debug modes are also available here (for example, **Normal** and **Albedo**).
EV (Exposure Value)	Use the slider to define the **Exposure Value** on the **HDRI** buffer before tone mapping.
Environment	Use the slider to choose which environment the view should use from the list in the HDRI view.
Rotation	Use the slider to rotate the environment around the **GameObject**. This rotation is added to the offset specific to each environment.
LoD or LoD__ (auto)__	Available only if the **Prefab** supports LOD. Use this slider to control level of detail (LOD) for the GameObject. Set it to –1 for automatic LOD (auto) selection.

If a multi-view mode is enabled, both views have their own settings. This is indicated by orange or blue to match the screen they represent.

Views menu in multi-view mode

HDRI environments in Look Dev

Leave feedback

Lighting in Look Dev is represented by an **HDRI** (high dynamic range image). The Look Dev view allows you to manipulate and easily switch between HDRIs to simulate different environments for the **Asset** you are working on.

This section provides information on how to create, import, browse and manipulate HDRI environments. See the following pages for details:

The HDRI view: The viewing panel and library for importing, browsing and using HDRIs as environments in Look Dev.
The HDRI menu: The settings and menus for configuring and manipulating HDRI environments.
Environment shadows: Guidance on configuring lighting and shadow in your HDRI environment.

The HDRI view

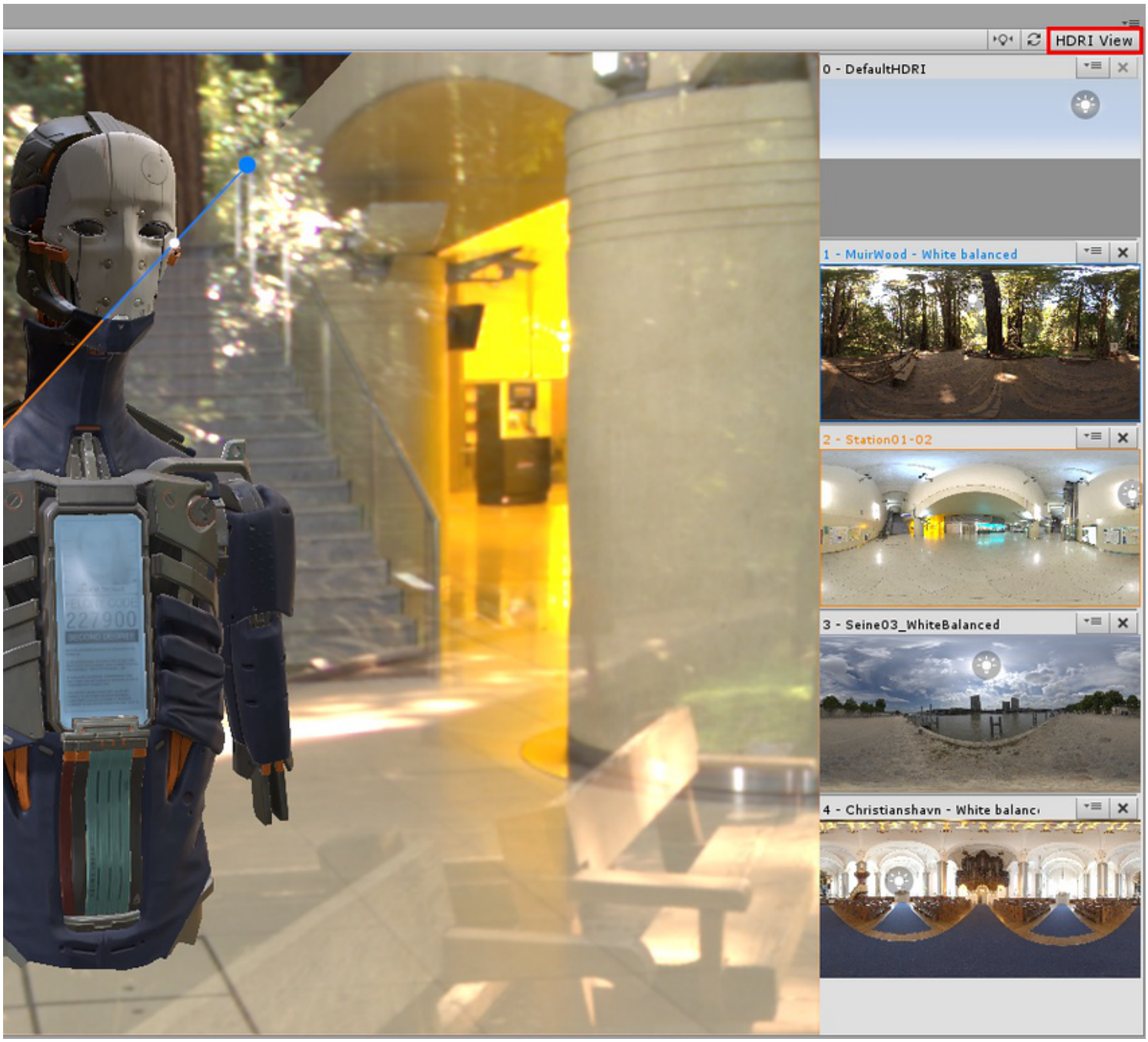
Leave feedback

Lighting in Look Dev is represented by an **HDRI** (high dynamic range image). The Look Dev view allows you to manipulate and easily switch between HDRIs.

The HDRI view allows you to view and browse all HDRIs in the HDRI Library. By default, the HDRI Library is saved within the Editor. You can also save the HDRI Library as an Asset under a different name. In the Look Dev, go to **Settings** > **HDRI Library** > **Save As New Library**.

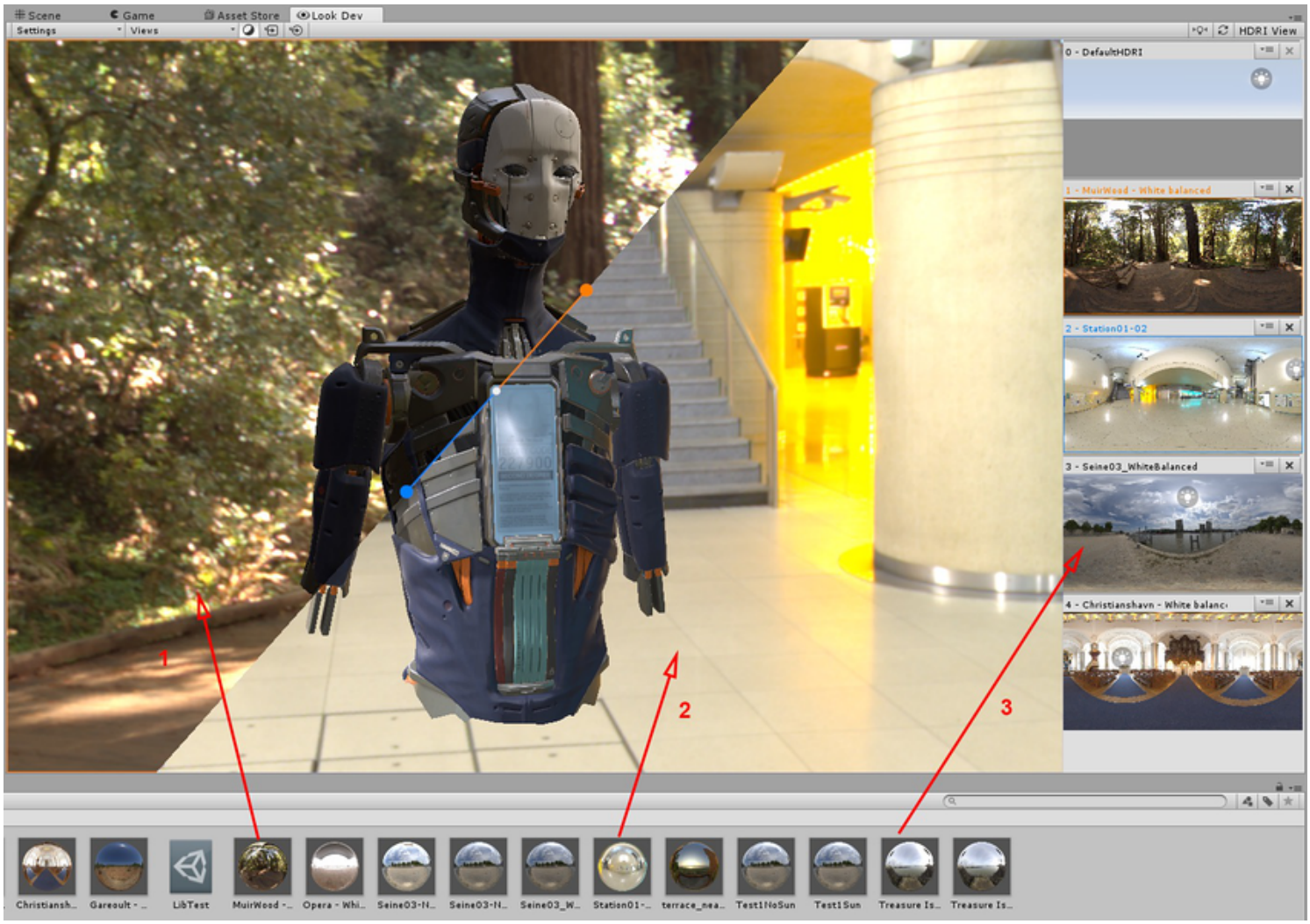
To import an HDRI into Unity, load a **.hdr** or .exr file into your Unity project like you would any other image. In the Texture Importer **Inspector** window, set **Texture Type** to **Default**, set **Texture Shape** to **Cube**, and set **Convolution Type** to **Specular (Glossy Reflection)**.

To browse the HDRI Library, select the **HDRI View** button in the top-right corner of the Look Dev view. This opens the HDRI view, which shows everything in the HDRI Library as seen in the image below.



Select the **HDRI View** button to see the HDRI Library

When you want to test a HDRI Texture **Asset** or a **skybox** cube map Material, drag and drop it into the Look Dev view or the HDRI view to use it. All HDRIs added to the Look Dev view or HDRI view are automatically saved to the HDRI Library. A blue or orange rectangle appears around the view while you do this to indicate which viewing panel it is to be assigned to when you drop it.



Drag and drop to test in the Look Dev view or HDRI view

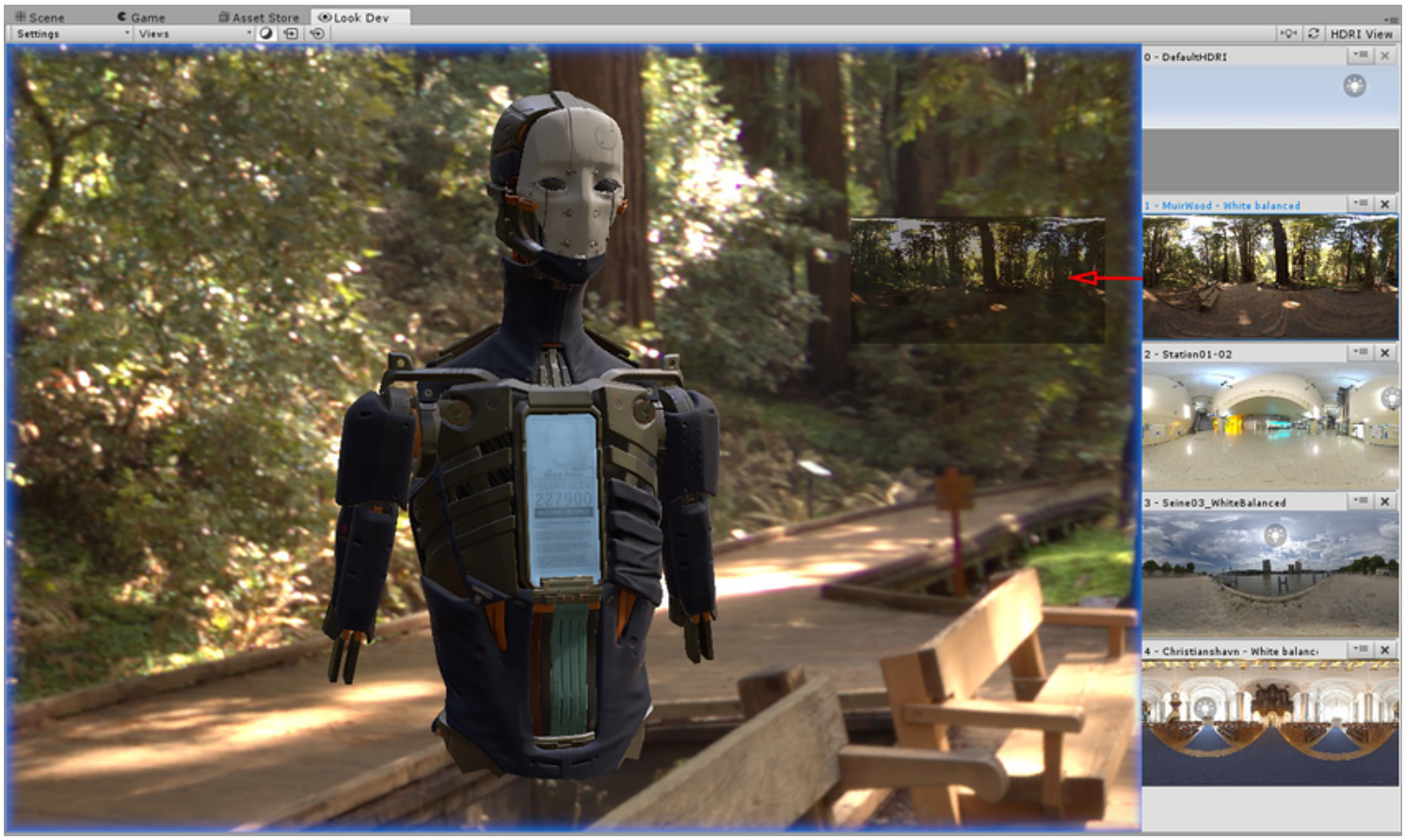
Image key:

Drag to orange view

Drag to blue view

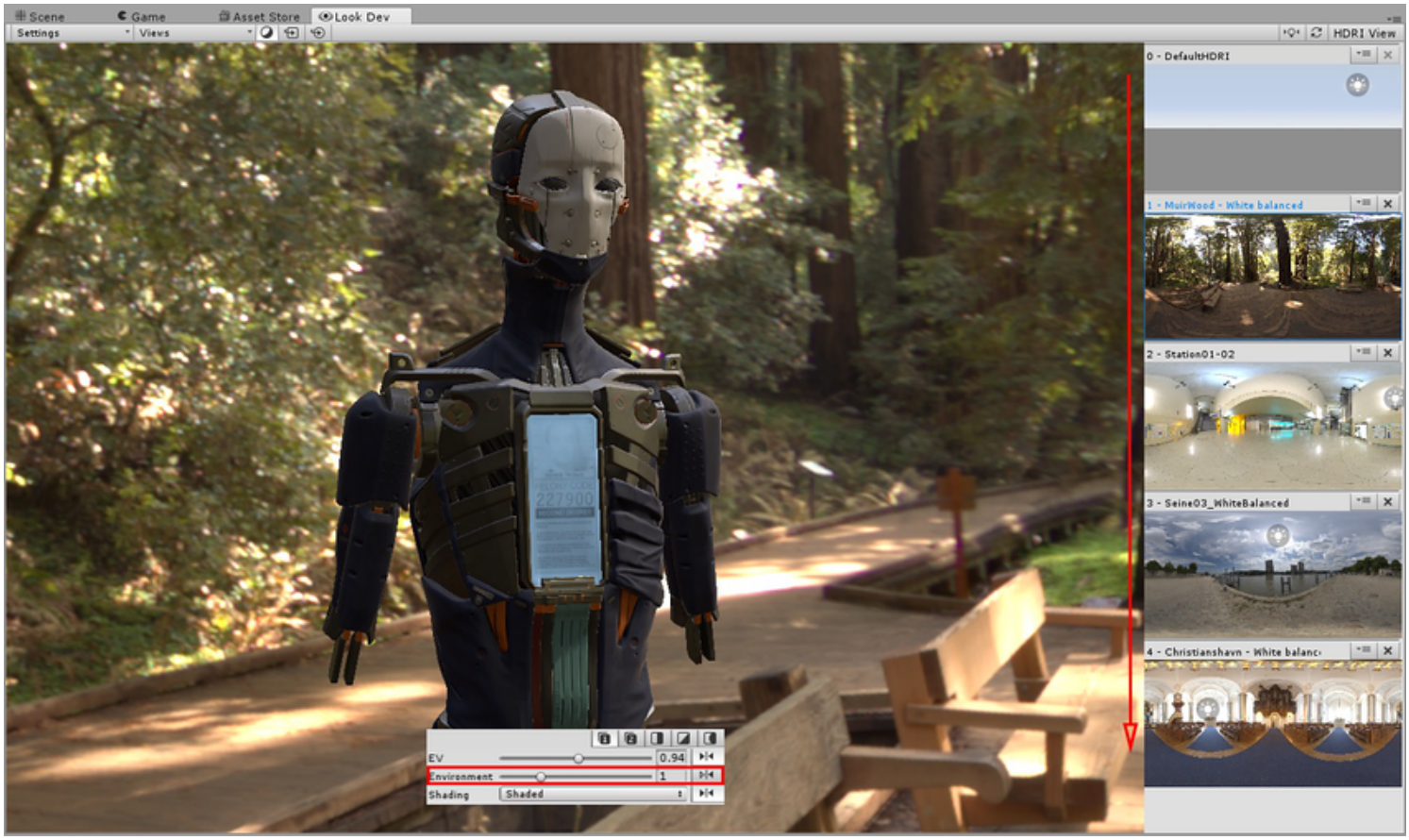
Drag to HDRI view. To drag and drop multiple HDRIs at once, hold down **Shift** while selecting them.

Drag and drop HDRIs straight from the HDRI view to move them into the Look Dev view.



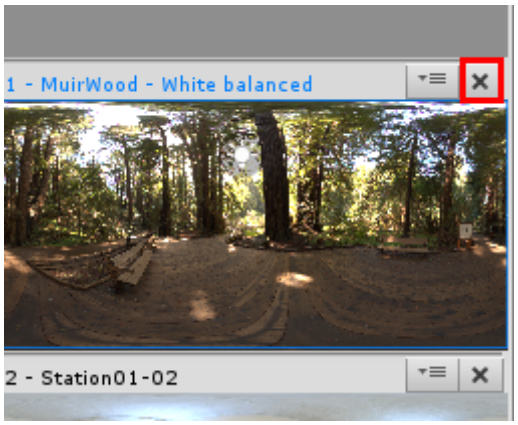
Drag and drop from the HDRI view to the Look Dev view

Use the **Environment** slider in the Control Panel to go back and forth through the HDRI list. The number displayed alongside the slider indicates which HDRI is currently being displayed, beginning at 0.



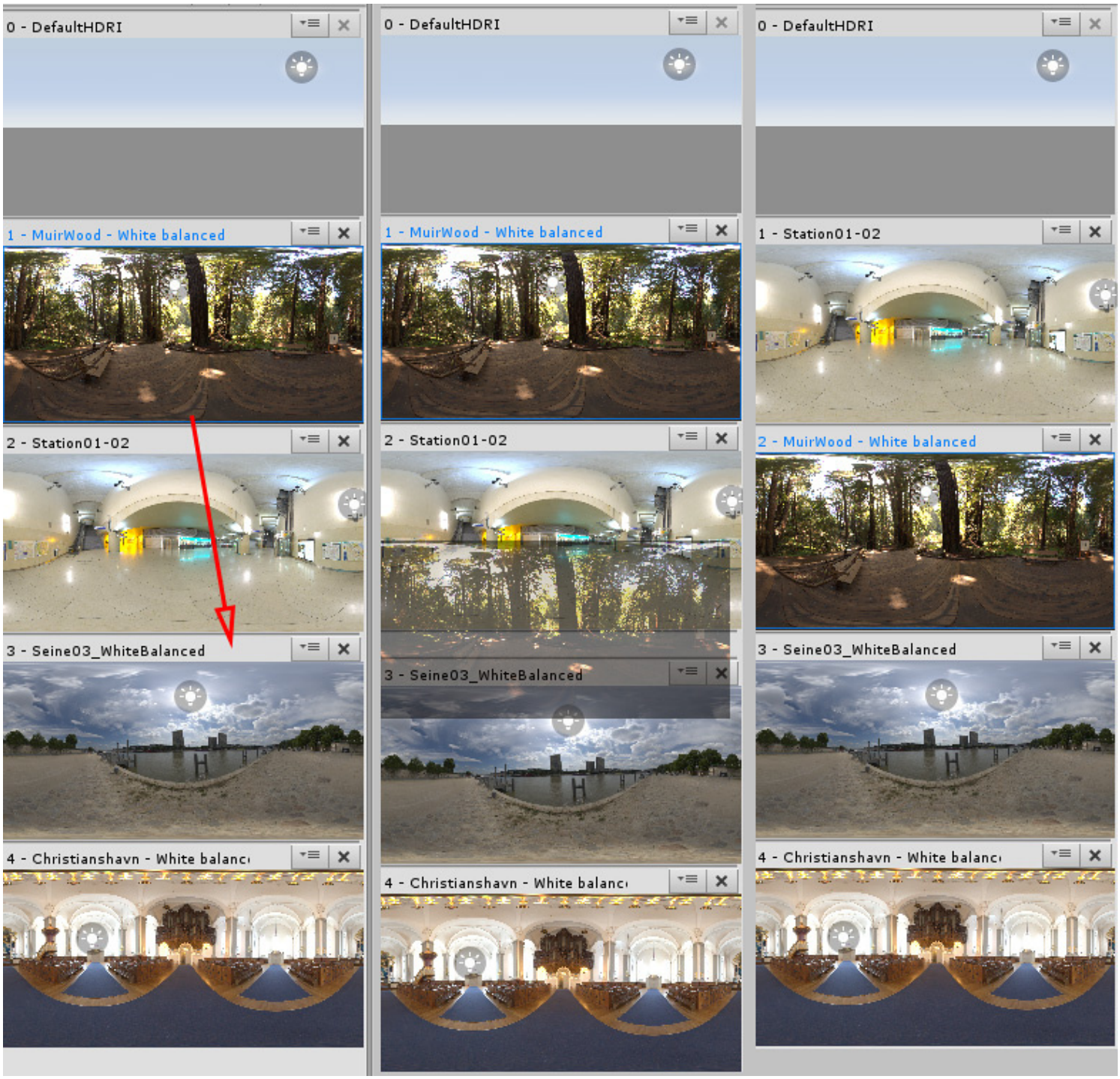
Using the **Environment** slider to scroll through the HDRI list

To remove an HDRI from the HDRI Library, click the **X** in the top right next to the HDRI's name. Note that you can't remove **DefaultHDRI**.



Click the **X** to remove an HDRI from the HDRI Library

To reorganize the HDRI view, drag and drop HDRIs inside the browser itself.



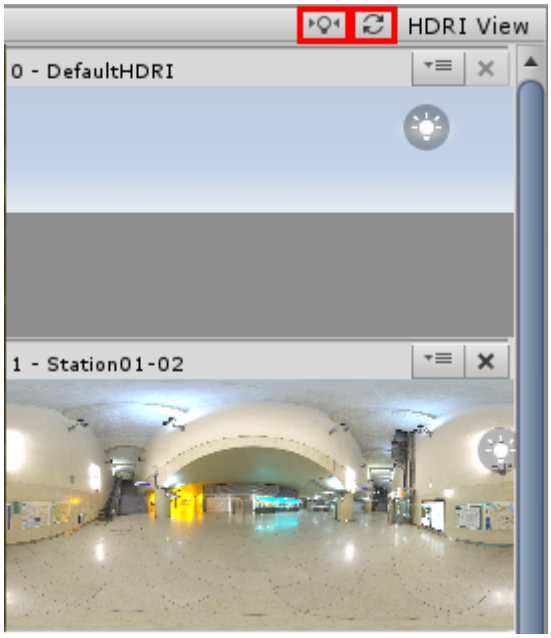
Drag and drop to reogranise the HDRI view

HDRI menus

Leave feedback

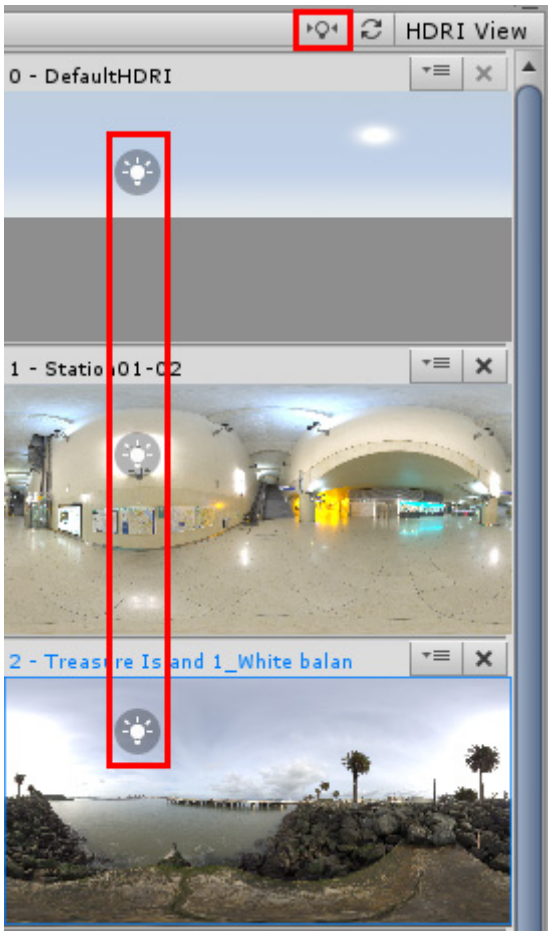
HDRI main settings

Use these settings to apply lighting and shadow changes to all **HDRIs** in the HDRI view with one action.



Main setting buttons

Use the lightbulb button to synchronise light placement vertically across all HDRI environments. This is useful if you are switching from one environment to another and you want to keep the shadow direction similar for comparison.



The lighbulb button synchronises light placement

: Use this button to reset the lighting position of all HDRIs.

HDRI individual settings

Click on the 3-lines button (highlighted here) to access each HDRI's properties menu

Each HDRI has its own menu to control its properties. To access this, click on the three-lines button (a representation of three sliders) underneath the **Environment** button.

Environment

Menu Item	Properties
Angle Offset	Give a number by which to offset the angle of the HDRI (shortcut: left click+Ctrl on the HDRI thumbnail in the HDRI View). This offset is unique to the HDRI selected and is different from the **Rotation** slider in the Look Dev Views menu, which works independently from the HDRI offset.

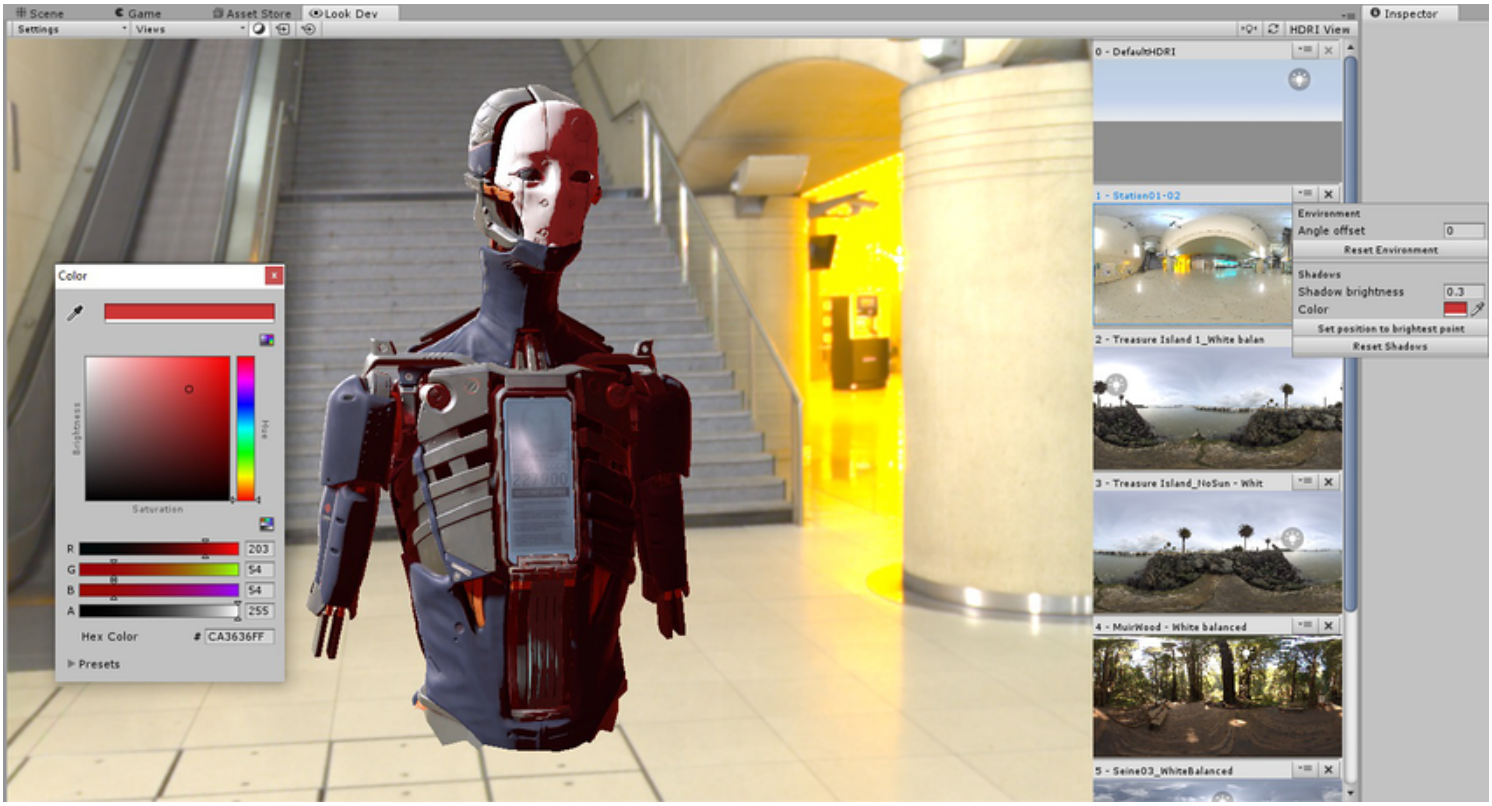
Menu Item	Properties
Reset Environment	Resets the Angle offset to default (0).

An environment in Look Dev with two different angle offsets applied to it

Shadows

This applies only if **Environment Shadow** is enabled.

Menu item	Properties
Shadow brightness	Controls the brightness of the shadow.
Color	Controls the color of the shadow. This is useful to match particular lighting conditions (for example if the lighting is all blue).
Set to brightest	Sets the position of the shadow direction to the brightest texel of the HDRI thumbnail.
Reset Shadows	Resets all shadow controls to default.



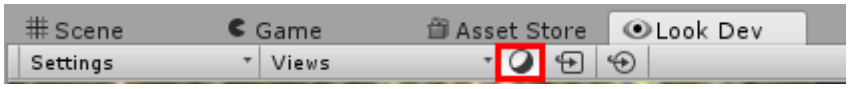
The **Shadows** menu

Environment Shadow

Leave feedback

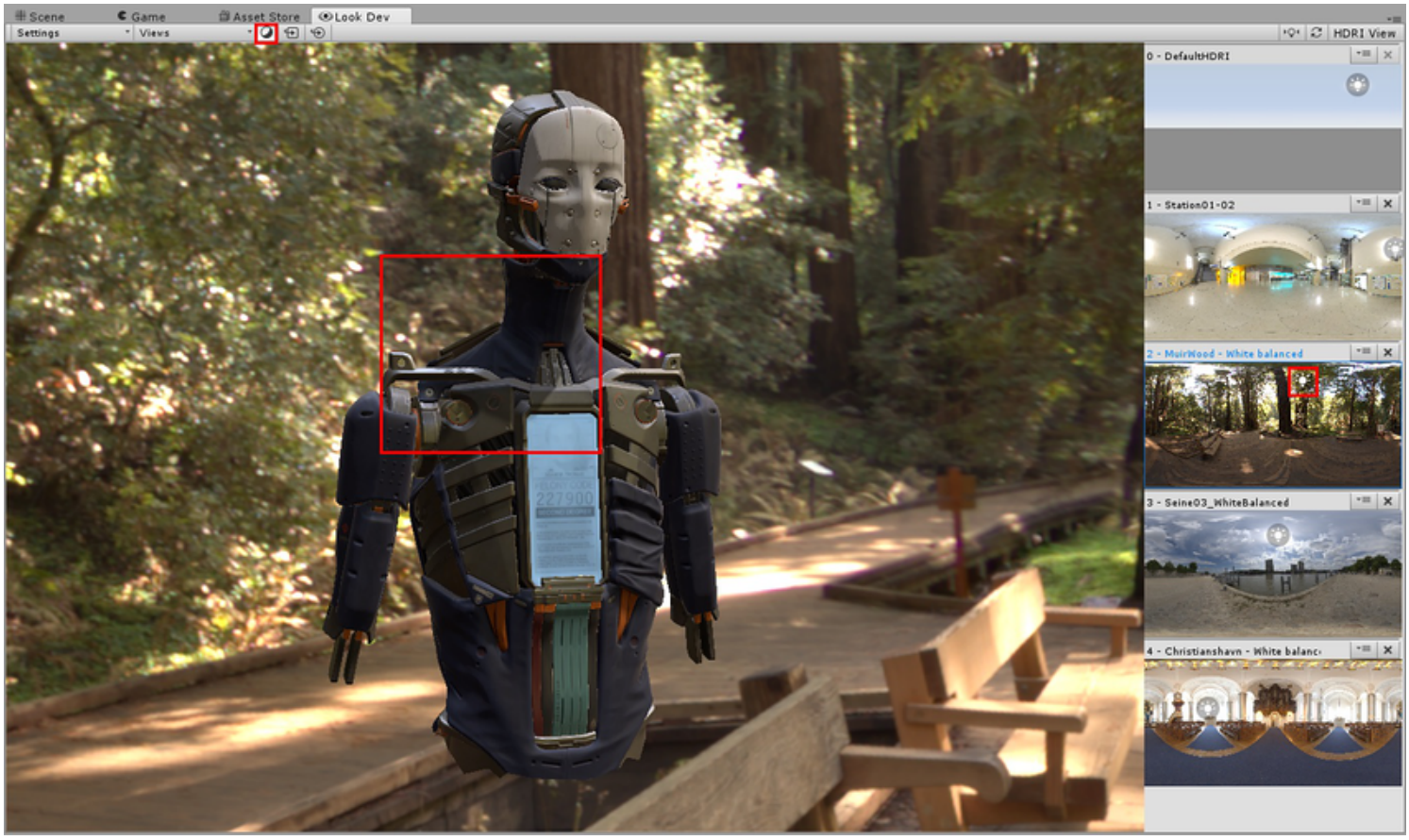
You can use Look Dev to simulate environmental shadows. Note that this simulation is not completely accurate, and only shows harsh shadows.

To simulate environmental shadows, click the **Environment Shadow** button in the top menu of the Look Dev window (highlighted in the image below).



The top menu of the Look Dev window with the **Environment Shadow** button highlighted

Use the **HDRI** view to control the direction of the shadows. When you toggle the **Environment Shadow** on, a sun icon appears over the selected **HDRI** in the browser. Click and drag this to orient the shadow, and hold Shift+left click to manipulate the shadow direction.



Manipulate the sun icon (here in the middle pane on the right-hand screen) to control shadow direction

Environment Shadow works best when there is a key light source in the HDRI (like the Sun, or a lamp). To make that a light source, drag and drop the sun icon on top of it. By default, the light source is set up on the brightest texel of the HDRI. For more evenly distributed light sources (like an overcast sky), this might not give accurate or expected results.

When **Environment Shadow** is enabled, Look Dev uses one HDRI for the **Prefab** when it is lit by the key light source (this is called the **Lit HDRI**), and one for the Prefab when it is in the shadow of the key light source (this is called the **Shadow HDRI**). The shadow in the **Shadow HDRI** is based on the light direction from the light icon. By default, the **Shadow HDRI** is the same as the **Lit HDRI**, but with reduced luminosity.

To get more accurate results, provide your own **Shadow HDRI**. For best results, make this the same as the **Lit HDRI** but without the key light source (for example, by removing the Sun from the image).

To pair up a **Shadow HDRI** and a **Lit HDRI**, drag and drop the **Shadow HDRI** onto the **Lit HDRI**. The **Shadow HDRI** appears over the **Lit HDRI** with a moon icon, as shown in the image below.

Pairing up a **Shadow HDRI** and a **Lit HDRI**

When **Environment Shadow** is enabled, the **Lit HDRI** switches to the assigned **Shadow HDRI**.

To remove the **Shadow HDRI**, click on the cross.

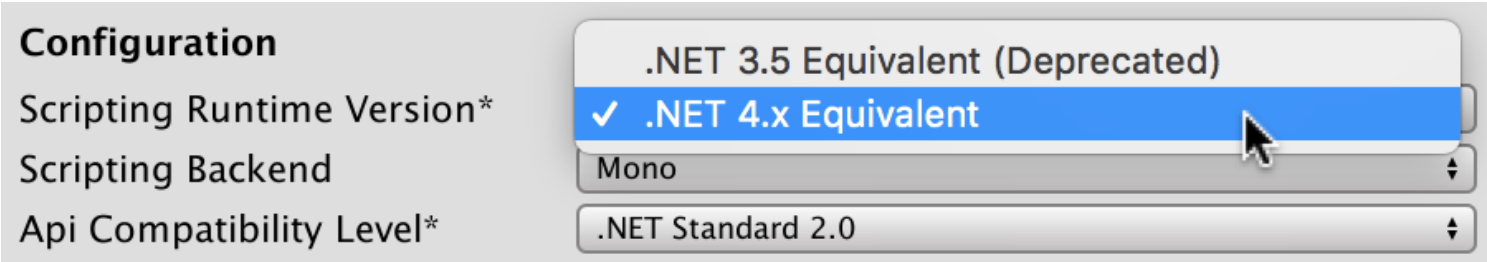
Click on the cross to remove the **Shadow HDRI**

Note: You can manipulate **Shadow HDRI** like any other HDRI in the library.

Scripting Runtime

Leave feedback

The default **Scripting Runtime Version** is .NET 4.6. (.NET 3.5 is marked as deprecated.) This option is a per-Project setting that you specify in the Unity Editor Player Settings (**Edit > Project Settings > Player**), in the **Configuration** section:



The equivalent scripting API is the **PlayerSettings.scriptingRuntimeVersion** property. **Important**: Changing this property requires an Editor restart because it affects the Editor as well as Players.

By default nothing should change about how Unity behaves or what .NET functionality is available. Once you use .NET 4.6 for a project, you will be able to use C# 6, .NET 4.6 class libraries, and new runtime features in user **scripts** and precompiled assemblies.

Note The .Net 3.5 scripting runtime is deprecated, but it will be available for Unity 2018.3 and 2018.4 LTS. Please migrate or start new Projects with the .NET 4.x runtime.

FAQ

What platforms does this affect?

All of them, but in different ways:

Editor and Standalone use the new version of Mono when this option is enabled.

All console platforms will only be able to use **IL2CPP** when targeting the new .NET version.

iOS and **WebGL** will continue to be IL2CPP only.

Android will continue to support both Mono and IL2CPP.

Other platforms are still undergoing work to support either new Mono or IL2CPP

What about IL2CPP?

IL2CPP fully supports the new .NET 4.6 APIs and features.

How stable is the updated Mono/IL2CPP?

The internal, automated tests for Unity pass with the new Mono/IL2CPP. Of course, we still expect you may encounter issues. Please file bugs for any issues you encounter.

Does this preview include a new GC?

No. This is an upgrade of the class libraries and runtime, but we are still using the Boehm GC. We are targeting a new version of Boehm that we use with IL2CPP (so IL2CPP and Mono will have exact same GC).

Wait, why don't we have a new GC?

The newer Mono garbage collector (SGen) requires additional work in Unity and will follow once we have stabilized the new runtime and class libraries.

Can I debug managed code with this new Mono?

VSTU 3.1 is required for our new Mono. Please install it to use the new Mono runtime on Windows.

Why are my builds larger with the new .NET version?

The .NET 4.6 class libraries are quite larger than our current .NET 3.5 class libraries. We are actively working on improving the managed linker to reduce size further.

Additionally, we are working on a new Unity specific class library API profile (like our current **unity** profile) that will:

a) be specifically implemented to work on AOT platforms

b) smaller in surface area and internally designed to be strippable/linkable

c) support netstandard 2.0 (yet to be officially released)

When I try this new option, something doesn't work. What should I do?

Please file a bug. We will be rapidly fixing them!

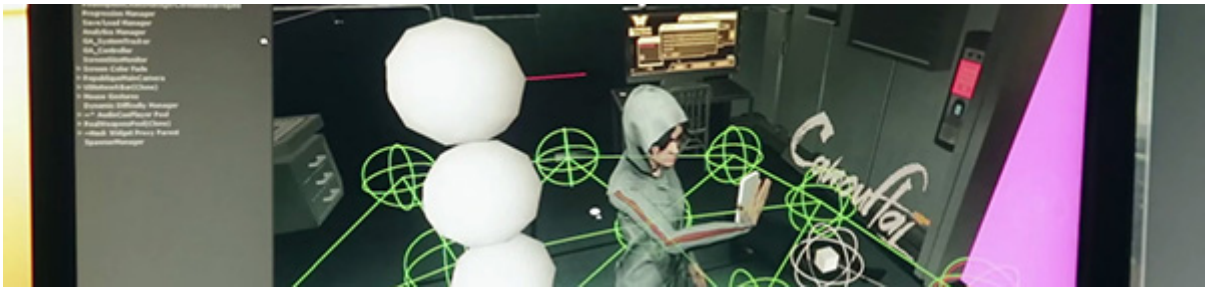
2018–08–02 Page published with no editorial review

MonoDevelop replaced by Visual Studio from 2018.1

.Net 3.5 scripting runtime deprecated in Unity 2018.3

Legacy Topics

Leave feedback



Find information here on historical functionality. This is not supported in the current Unity version.

Windows Store: Windows SDKs

Leave feedback

WARNING: LEGACY DOCUMENTATION

Note that that from 2017.1 this documentation is out-dated. 2017–06–30

Unity allows you to target these Windows SDKs when targeting Windows Store:

Windows 8.1 (Runs on Windows 8.1 and newer)
Windows Phone 8.1 (Runs on Windows Phone 8.1 and newer)
Universal 8.1 (Runs on both Windows and Windows Phone 8.1 or newer)
Universal 10 (Runs on all Windows 10 or newer devices that are part of Universal Windows Platform)

2017–06–30 Page amended with no editorial review

Windows Phone 8.1

Leave feedback

WARNING: LEGACY DOCUMENTATION

Note that that from Unity 2017.1 this documentation is out-dated. 2017–06–30

Windows Phone 8.1 is the latest edition of Microsoft's Windows Phone mobile operating system, succeeding Windows Phone 8. Windows Phone 8.1 focuses a lot on convergence between Windows Phone and Windows Store platforms into a single developer platform that runs the same types of apps. Windows Phone 8.1 and Windows Store share a similar app model and life cycle, a shared toolset, a common XAML **UI** framework and CLR runtime (which replaced Silverlight, used in Windows Phone 8.0). Windows Phone 8.1 uses AppX packaging for applications, rather than the old XAP package.

Windows Phone 8.1 introduces a new notification center, along with a set of capabilities developers can use to manage toast notifications through local API calls or by using push. Through these APIs and mechanisms, an app can manage the toast notifications they send to users. Operations include add, remove, replace, and expire. All notification classes are in the Windows.UI.Notifications namespace.

Windows Phone 8.1 supports many more DirectX features when compared to Windows Phone 8. These include:

Direct2D APIs DirectWrite APIs Windows Imaging **Component** APIs Runtime **shader** compilation and reflection GPU access to YUV shaders GPU performance counters Graphics diagnostics support Windows Phone 8.1 has adopted the SwapChainPanel XAML control in favor of BackgroundDrawingSurfaceGrid, which was used on Windows Phone 8.

Windows Phone 8.1 provides APIs that you can use to create an app that records images or videos of the phone's screen activity. This way games can upload and share in-game activities. This feature is enabled through enhancements to the Windows.Media.Capture APIs already available to Windows Store apps.

Windows Phone 8.1 emulator adds support for SD card emulation, and multi-touch with mouse, improved **networking** support to simplify setup, configuration, and access to network resources and to reduce troubleshooting.

Windows Phone 8.1 adds support for remote performance monitoring, ETW logging, and App Verifier. Existing Windows Phone 8 command-line tools – Isolated Storage Explorer, the deployment tool, and the phone registration tool – are still supported. The Simulation Dashboard is not updated for Windows Phone (code-named "Blue").

2017–06–30 Page amended with no editorial review

Windows Phone 8.1: Debugging

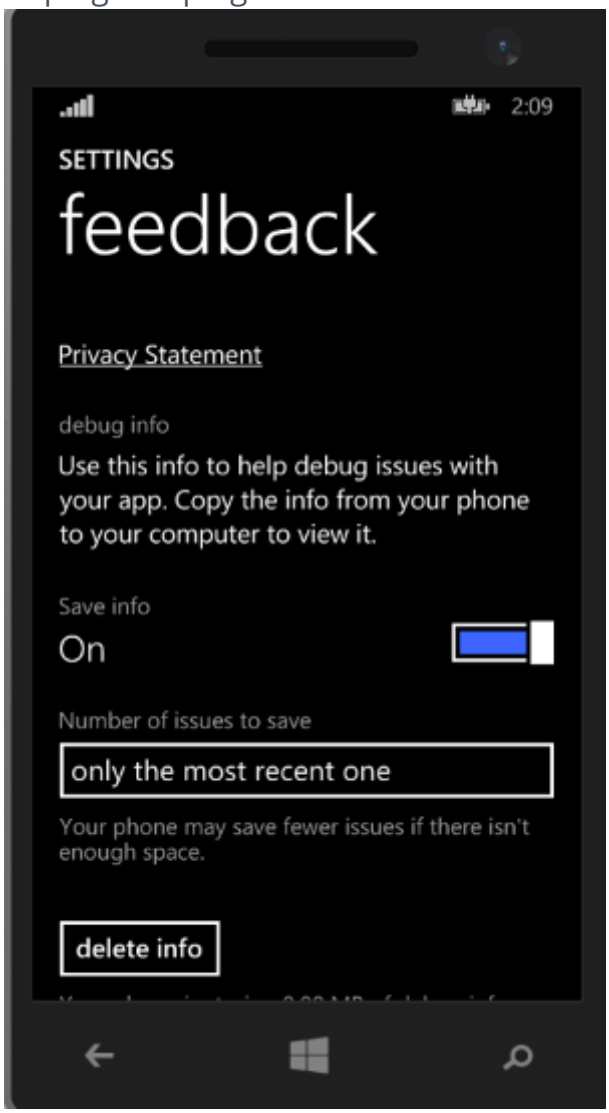
Leave feedback

WARNING: LEGACY DOCUMENTATION

Note that that from Unity 2017.1 this documentation is out-dated. 2017–06–30

When the application crashes without the debugger attached, it is possible to get a dump file, if appropriate settings are enabled on the phone.

Go to Settings->Feedback
Enable **Save Info** to **On**
When the application crashes, the dump file should be saved in *This PC<Phone Name>\Phone\Documents\Debug*
Note: When the dump file is saved, windows explorer might still show an empty folder, in that case unplug and plug USB cable to force windows explorer to refresh that folder.



See Also Windows Store Apps debugging, log files

2017–06–30 Page amended with no editorial review

Windows 8.1 Universal Applications

Leave feedback

WARNING: LEGACY DOCUMENTATION

Note that that from Unity 2017.1 this documentation is out-dated. 2017–06–30

Windows 8.1 Universal Applications is a way to target both Windows Store and Windows Phone target with a - single Visual Studio project that works on both Windows 8.1 and Windows Phone 8.1: desktops, laptops, tablets and phones. It is a direct result of platform convergence movement.

You can read more about Windows 8.1 Universal Applications here: http://dev.windows.com/en-us/develop/Building-universal-Windows-apps

Unity exposes a way to build Universal Windows Store/Phone 8.1 applications, through selecting “Universal 8.1” SDK in build window. Upon building such a project, Unity creates a Universal Visual Studio project, which then can be built to both Windows and Windows Phone devices.

How does it work?

Windows Phone 8.1 and Windows 8.1 are still not binary compatible - you cannot run a single DLL on both platforms unless it is a portable class library, which would mean you would not be able to access platform specific APIs on them (like SMS API on the Windows Phone and mouse API on Windows). Therefore, we compile two versions of assemblies.

There are two main differences between assemblies compiled for the phone and the store: preprocessor directives and target SDK. Windows gets to target Windows .NET Core, while the Phone targets Phone .NET Core. They’re almost identical, although there are some very minor differences.

Universal Project folder structure looks like this:

```
UniversalApp1 - (solution directory)
    UniversalApp1.Windows - (here goes windows specific files, all Windows DLLs)
        -
        -
        -
    UniversalApp1.WindowsPhone - (here goes windows phone specific files, all Wi
        -
        -
        -
    UniversalApp1.Shared - (here goes shared files)
        -
        -
        -
```

When you build a Universal Application, you will produce two binaries out of single project - one for Windows and one for Windows Phone. Neither AppX package has redundant files left from over the other platform - all thanks

to the project structure.

2017–06–30 Page amended with no editorial review

Universal Windows 10 Applications: Getting Started

Leave feedback

WARNING: LEGACY DOCUMENTATION

Note that that from Unity 2017.1 this documentation is out-dated. 2017–06–30

Getting Started

Currently if you want to build for Universal Windows 10 Applications (a.k.a UWP), you have to do it on Windows 10.

Unity supports three UWP: X86, X64 and ARM.

Prerequisites

In order to build applications for Universal Windows 10 application, Visual Studio 2015 RTM installation is required, the minimum version is 14.0.23107.0

2017–06–30 Page amended with no editorial review

Asset Server (Team License)

Leave feedback

WARNING: LEGACY DOCUMENTATION

Note that that from 2017.1 the **Asset** Server a deprecated product. We recommend using Plastic SCM or Perforce for version control in your Unity project. 2017–06–30

Unity Asset Server Overview

PlasticSCM PerForce

The **Unity Asset Server** is an asset and version control system with a graphical user interface integrated into Unity. It is meant to be used by team members working together on a project on different computers either in-person or remotely. The Asset Server is highly optimized for handling large binary assets in order to cope with large multi gigabyte project folders. When uploading assets, **Import Settings** and other meta data about each asset is uploaded to the asset server as well. Renaming and moving files is at the core of the system and well supported.

It is available only for Team License users. To purchase Team License (if you do not have it as part of Unity Pro) please visit the Unity store at http://unity3d.com/store

Note that Asset Server is a legacy product and is no longer maintained.

New to Source Control?

If you have never used Source Control before, it can be a little unfriendly to get started with any versioning system. Source Control works by storing an entire collection of all your assets - meshes, textures, materials, **scripts**, and everything else - in a database on some kind of server. That server might be your home computer, the same one that you use to run Unity. It might be a different computer in your local network. It might be a remote machine colocated in a different part of the world. It could even be a virtual machine. There are a lot of options, but the location of the server doesn't matter at all. The important thing is that you can access it somehow over your network, and that it stores your game data.

In a way, the Asset Server functions as a backup of your Project Folder. You do not directly manipulate the contents of the Asset Server while you are developing. You make changes to your Project locally, then when you are done, you **Commit Changes** to the Project on the Server. This makes your local Project and the Asset Server Project identical.

Now, when your fellow developers make a change, the Asset Server is identical to their Project, but not yours. To synchronize your local Project, you request to **Update from Server**. Now, whatever changes your team members have made will be downloaded from the server to your local Project.

This is the basic workflow for using the Asset Server. In addition to this basic functionality, the Asset Server allows for rollback to previous versions of assets, detailed file comparison, merging two different scripts, resolving conflicts, and recovering deleted assets.

Setting up the Asset Server

The Asset Server requires a one time server setup and a client configuration for each user. You can read about how to do that in the Asset Server Setup page.

The rest of this guide explains how to deploy, administrate, and regularly use the Asset Server.

Daily use of the Asset Server

This section explains the common tasks, workflow and best practices for using the Asset Server on a day-to-day basis.

Getting Started

If you are joining a team that has a lot of work stored on the Asset Server already, this is the quickest way to get up and running correctly.

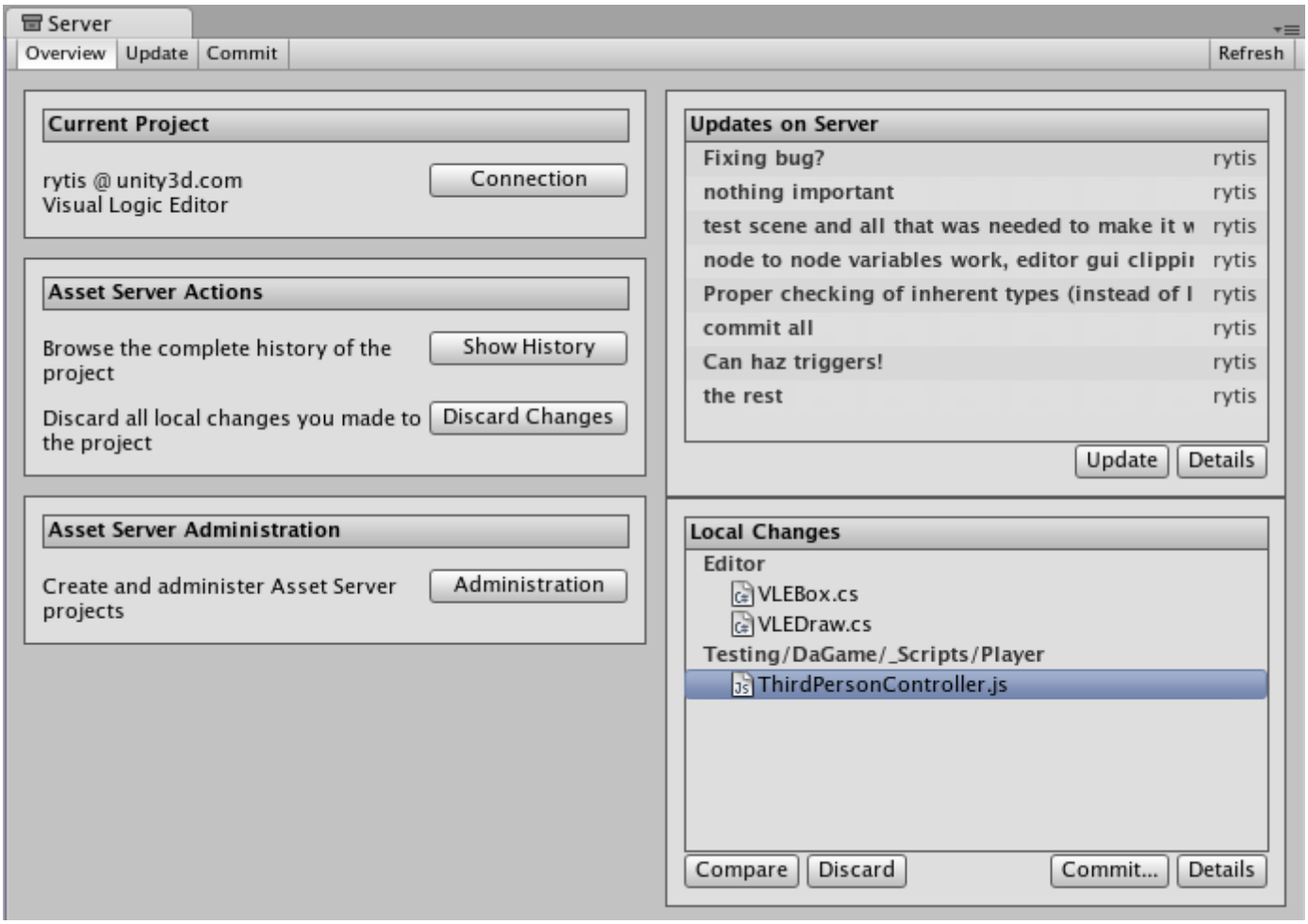
Create a new empty Project with no packages imported
Go to **Edit** > **Project Settings** > **Editor** and select **Asset Server** as the version control mode
From the menubar, select **Window** > **Asset Management** > **Version Control**
Click the **Connection** button
Enter your user name and password (provided by your Asset Server administrator)
Click **Show Projects** and select the desired project
Click **Connect**
Click the **Update** tab
Click the **Update** button
If a conflict occurs, discard all local versions
Wait for the update to complete
You are ready to go

Workflow Fundamentals

When using the Asset Server with a multi-person team, it is generally good practice to Update all changed assets from the server when you begin working, and Commit your changes at the end of the day, or whenever you're done working. You should also commit changes when you have made significant progress on something, even if it is in the middle of the day. Committing your changes regularly and frequently is recommended.

Understanding the Server View

The **Server View** is your window into the Asset Server you're connected to. You can open the Server View by selecting **Window** > **Asset Management** > **Version Control**.

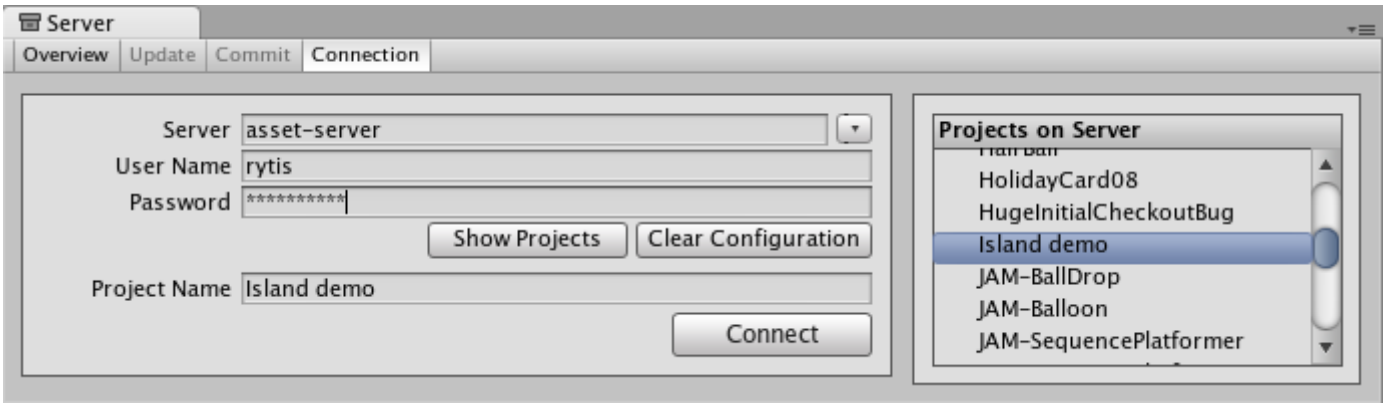


The **Overview** tab

The Server View is broken into tabs: **Overview Update**, and **Commit**. **Overview** will show you any differences between your local project and the latest version on the server with options to quickly commit local changes or download the latest updates. **Update** will show you the latest remote changes on the server and allow you to download them to your local project. **Commit** allows you to create a **Changeset** and commit it to the server for others to download.

Connecting to the server

Before you can use the asset server, you must connect to it. To do this you click the **Connection** button, which takes you to the connection screen:



The Asset Server connection screen

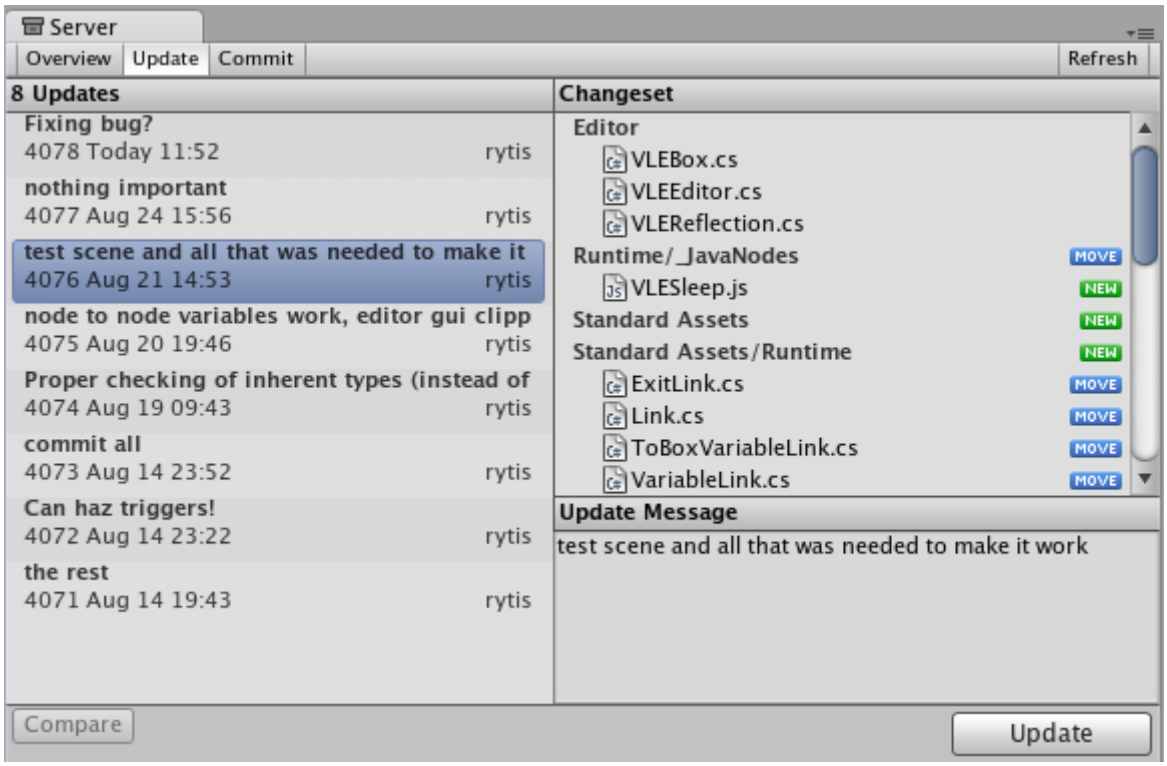
Here you need to fill in:

Server address
Username
Password

By clicking **Show projects** you can now see the available projects on the asset server, and choose which one to connect to by clicking **Connect**. Note that the username and password you use can be obtain from your system administrator. Your system administrator created accounts when they installed Asset Server.

Updating from the Server

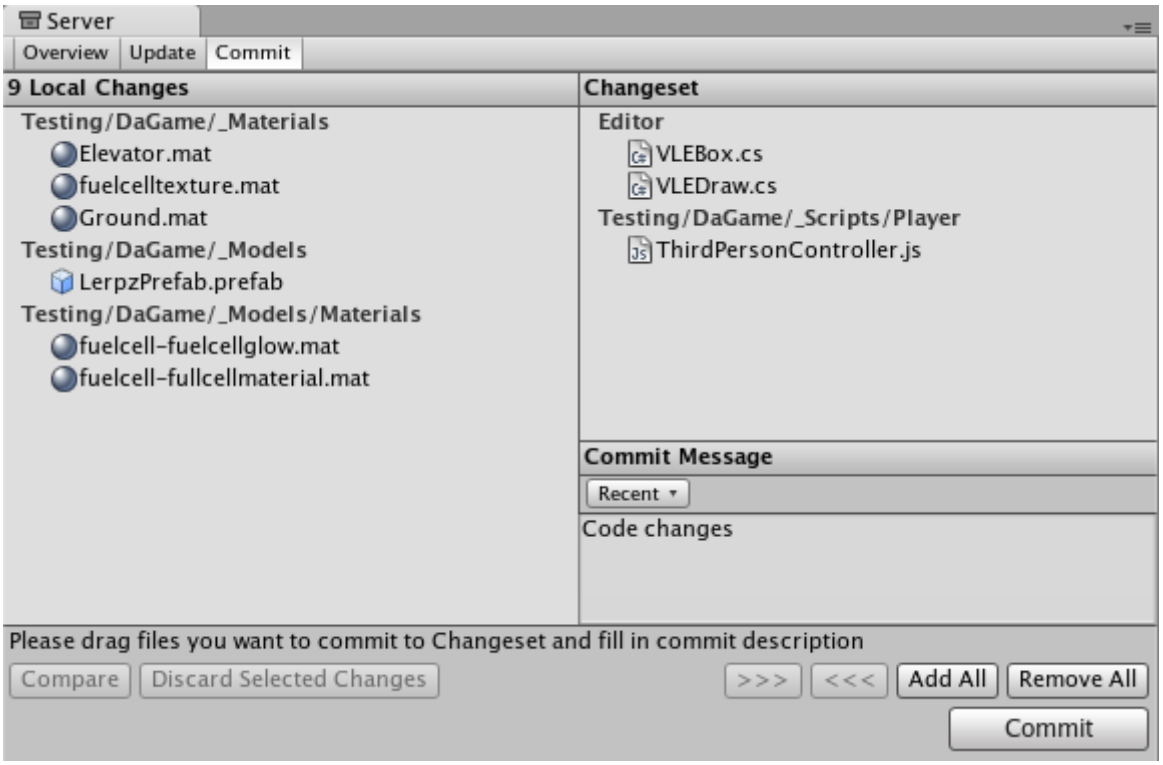
To download all updates from the server, select the **Update** tab from the Overview tab and you will see a list of the latest committed Changesets. By selecting one of these you can see what was changed in the project as well as the provided commit message. Click **Update** and you will begin downloading all Changeset updates.



The **Update** Tab

Committing Changes to the Server

When you have made a change to your local project and you want to store those changes on the server, you use the top **Commit** tab.

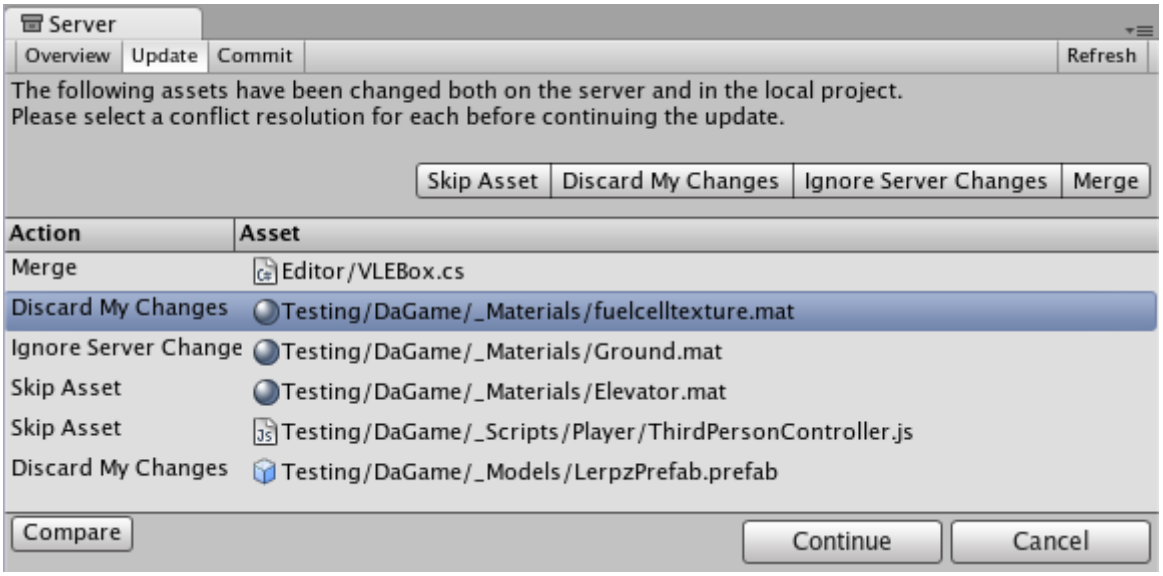


The **Commit** tab

Now you will be able to see all the local changes made to the project since your last update, and will be able to select which changes you wish to upload to the server. You can add changes to the changeset either by manually dragging them into the changeset field, or by using the buttons placed below the commit message field. Remember to type in a commit message which will help you when you compare versions or revert to an earlier version later on, both of which are discussed below.

Resolving conflicts

With multiple people working on the same collection of data, conflicts will inevitably arise. Remember, there is no need to panic! If a conflict exists, you will be presented with the **Conflict Resolution** dialog when updating your project.



The **Conflict Resolution** screen

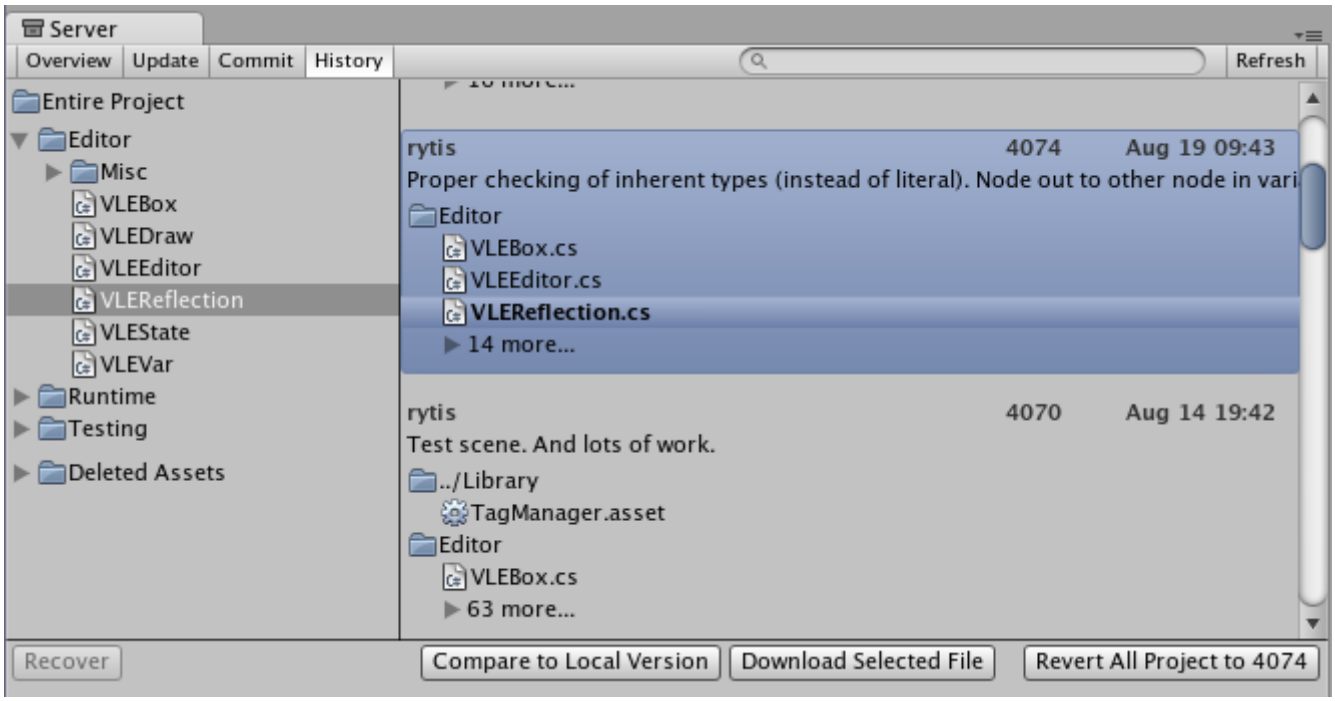
Here, you will be informed of each individual conflict, and be presented with different options to resolve each individual conflict. For any single conflict, you can select **Skip Asset** (which will not download that asset from the server), **Discard My Changes** (which will completely overwrite your local version of the asset) or **Ignore Server Changes** (which will ignore the changes others made to the asset and after this update you will be able to commit your local changes over server ones) for each individual conflict. Additionally, you can select **Merge** for text assets like scripts to merge the server version with the local version.

Note: If you choose to discard your changes, the asset will be updated to the latest version from the server (i.e., it will incorporate other users' changes that have been made while you were working). If you want to get the asset back as it was when you started working, you should revert to the specific version that you checked out. (See *Browsing revision history and reverting assets* below.)

If you run into a conflict while you are committing your local changes, Unity will refuse to commit your changes and inform you that a conflict exists. To resolve the conflicts, select **Update**. Your local changes will not automatically be overwritten. At this point you will see the **Conflict Resolution** dialog, and can follow the instructions in the above paragraph.

Browsing revision history and reverting assets

The Asset Server retains all uploaded versions of an asset in its database, so you can revert your local version to an earlier version at any time. You can either select to restore the entire project or single files. To revert to an older version of an asset or a project, select the Overview tab then click **Show History** listed under Asset Server Actions. You will now see a list of all commits and be able to select and restore any file or all project to an older version.



The **History** dialog

Here, you can see the version number and added comments with each version of the asset or project. This is one reason why descriptive comments are helpful. Select any asset to see its history or **Entire Project** for all changes made in project. Find revision you need. You can either select whole revision or particular asset in revision. Then click **Download Selected File** to get your local asset replaced with a copy of the selected revision. **Revert All Project** will revert entire project to selected revision.

Prior to reverting, if there are any differences between your local version and the selected server version, those changes will be lost when the local version is reverted.

If you only want to abandon the changes made to the local copy, you don't have to revert. You can discard those local modifications by selecting **Discard Changes** in the main asset server window. This will immediately download the current version of the project from the server to your local Project.

Comparing asset versions

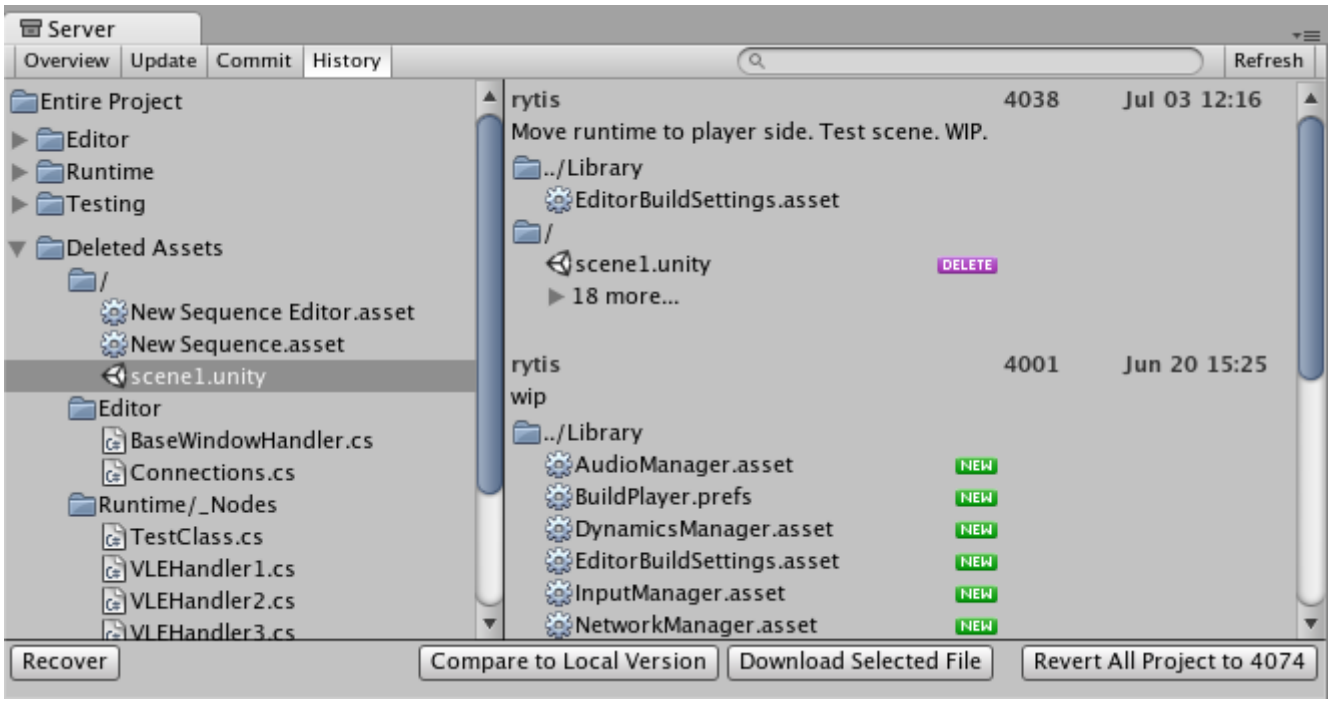
If you're curious to see the differences between two particular versions you can explicitly compare them. To do this, open **History** window, select revision and asset you want to compare and press **Compare to Local Version**. If you need to compare two different revisions of an asset - right click on it, in the context menu select **Compare to Another Revision** then find revision you want to compare to and select it.

Note: this feature requires that you have one of supported file diff/merge tools installed. Supported tools are:

On Windows:
TortoiseMerge: part of TortoiseSVN or a separate download from the project site.
WinMerge.
SourceGear Diff/Merge.
Perforce Merge (p4merge): part of Perforce's visual client suite (P4V).
TkDiff.
On Mac OS X:
SourceGear Diff/Merge.
FileMerge: part of Apple's XCode development tools.
TkDiff.
Perforce Merge (p4merge): part of Perforce's visual client suite (P4V).

Recovering deleted assets

Deleting a local asset and committing the delete to the server will in fact not delete an asset permanently. Just as any previous version of an asset can be restored through **History** window from the Overview tab.



The **History** dialog

Expand **Deleted Assets** item, find and select assets from the list and hit **Recover**, the selected assets will be downloaded and re-added to the local project. If the folder that the asset was located in before the deletion still exists, the asset will be restored to the original location, otherwise it will be added to the root of the Assets folder in the local project.

Best Practices & Common Issues

This is a compilation of best practices and solutions to problems which will help you when using the Asset Server:

Backup, Backup, Backup

Maintain a backup of your database. It is very important to do this. In the unfortunate case that you have a hardware problem, a virus, a user error, **etc** you may loose all of your work. Therefore make sure you have a backup system in place. You can find lots of resources online for setting up backup systems.

Stop the server before shutting the machine down

This can prevent "fast shutdowns" from being generated in the PostgreSQL (Asset Server) log. If this occurs the Asset Server has to do a recovery due to an improper shut down. This can take a very long time if you have a large project with many commits.

Resetting you password from Console

You can reset your password directly from a shell, console or command line using the following command:

psql -U unitysrv -d template1 -c"alter role admin with password 'MYPASSWORD'"

Can't connect to Asset Server

The password may have expired. Try resetting your password.

Also the username is case sensitive: "Admin" != "admin". Make sure you are using the correct case.

Make sure the server is actually running:

On OS X or Linux you can type on the terminal: ps -aux

On Windows you can use the Task Manager.

Verify that the Asset Server is not running on more than one computer in your Network. You could be connecting to the wrong one.

The Asset Server doesn't work in 64-bit Linux

The asset server can run OK on 64-bit Linux machines if you install 32-bit versions of the required packages. You can use "dpkg -i –force-architecture" to do this.

Use the Asset Server logs to get more information

Windows:

`ProgramFiles\Unity\AssetServer\log`

OS X:

`/Library/UnityAssetServer/log`

Asset Server training complete

You should now be equipped with the knowledge you need to start using the Asset Server effectively. Get to it, and don't forget the good workflow fundamentals. Commit changes often, and don't be afraid of losing anything.

2017–06–30 Page amended with no editorial review

Asset Server deprecated in 2017.1

Windows XP support in Standalone Player removed in 2018.1

Setting up the Asset Server

Leave feedback

WARNING: LEGACY DOCUMENTATION

Note that that from 2017.1 the **Asset** Server is a deprecated product. We recommend using Plastic SCM or Perforce for version control in your Unity project. 2017–06–30

Server-side Installation

The Asset Server is designed to be a simple one-time installation on a server machine. Interacting with the Asset Server is done through Unity. Unity can be installed on the server machine, but it does not need to be. It must be administrated from a Client machine, where Projects and Users can be added. Each additional client must be configured to synchronize with a Project, using a specific User credential.

You can install the Asset Server on Mac OS X 10.4 or later, Windows Vista and various Linux distributions including CentOS, Ubuntu and Suse Linux. Download Unity Asset Server from here.

The installer will install all necessary files, setup a database and launch the Asset Server. At the end of the process you will be asked to create an **Admin password**. This password is required to administer the Asset Server from within Unity. You must connect to the Asset Server as the administrator before you can create any projects or users.

Administrating the Asset Server

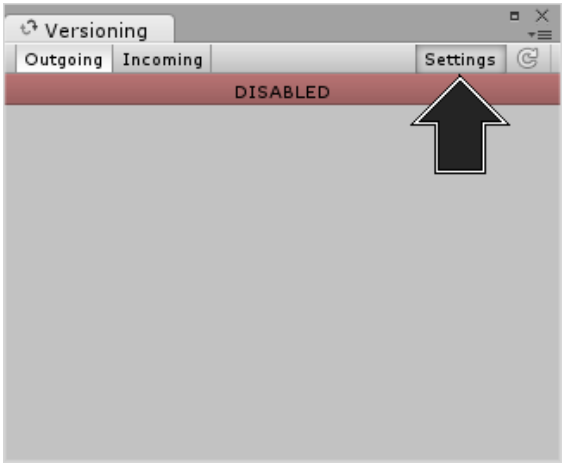
The Asset Server allows any number of **Users** to connect to a **Project**. The Administrator must first connect to the Server with Unity as a client and create new Projects and Users.

This guide will show you how to get your server up and running and ready for use.

Please be aware that the **Asset Server** is now a legacy product. We recommend using **Plastic SCM** or **Perforce** for version control in your Unity project.

PlasticSCM PerForce

1. Launch **Unity** and open up a blank project. Select **Window** > **Asset Management** > **Version Control**, then click the **Settings** button.



The **Versioning** tab

You will then see the **Version Control** settings in the **Inspector**, now select **Asset Server** from the **Mode** drop down menu.

The **Version Control Settings**

2. Now we are going to run the asset server. Find and run the following file:

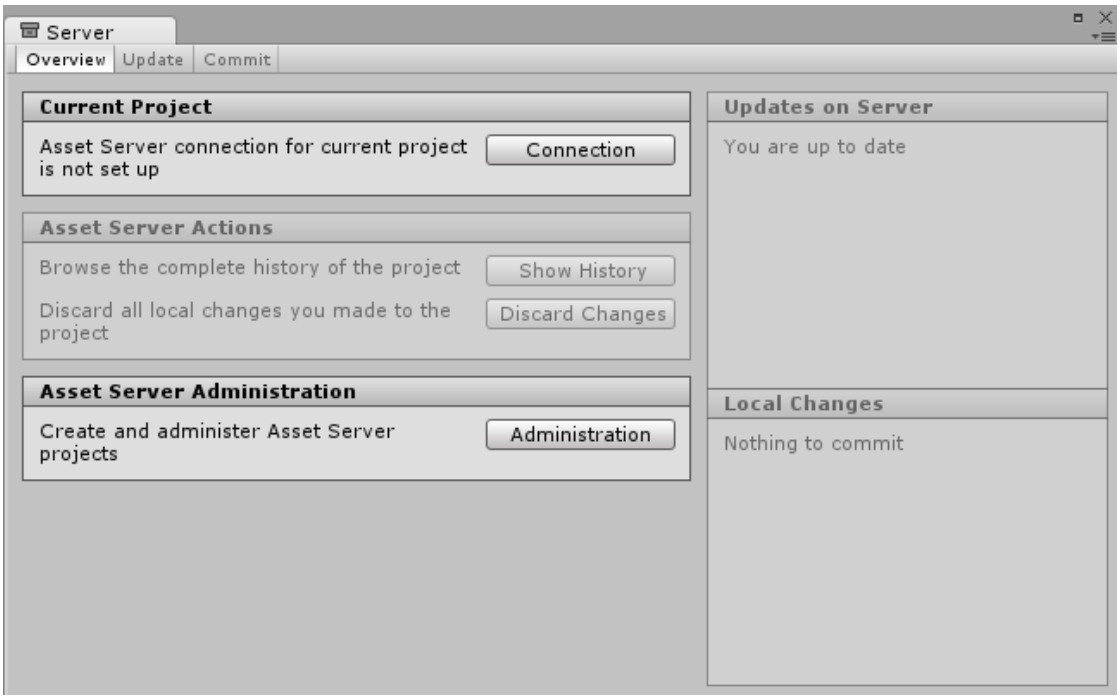
Windows: C:\Program Files (x86)\Unity\AssetServer\bin\AssetServerControl.exe

Macintosh: /Library/UnityAssetServer

Note: Yosemite 10.10.3 users will need to search via the hard drive's liabry directory, not the user's liabry directory.

3. You can now select **Window** > **Asset Management** > **Version Control**. This will open the asset server's management tab.

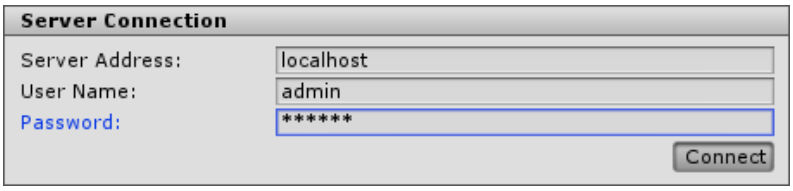
Click the Administration button.



The **Administration** tab

4. In the **Server Address** field, enter either the **ip address** or **host name** of the computer running the Asset Server that you want to administer. If the Asset Server is installed on your local machine, you can use "localhost" as the **Server Address**. Next, provide the

administrator name and password. The administrator name is always “admin”, and the password is what was entered when installing the Asset Server. Finally, hit the **Connect** button. You’re now connected to the Asset Server, and can perform the initial setup.

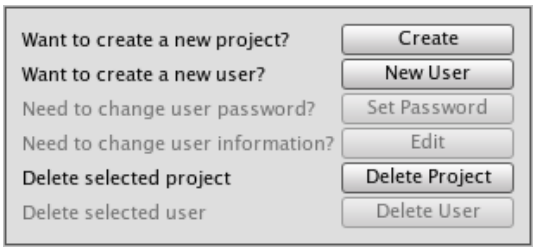


The **Administration** tab

Managing Projects and Users

Each Server can contain several Projects, and each User can have permission to one or more Projects. Projects are generally orthogonal, and unique in asset collections. It is best to think “one Project equals one game”.

New Projects can be created by clicking on the **Create** button in the **Server Administration** tab.



Click Create, then enter a name for the new project

New users can be created by first selecting an existing project and then clicking on the **New User** button.

Creating a new user

After a user has been created in one Project, the user can be added to another project by enabling the checkbox on the left of the user name in the users list.

You can enable or disable user access for individual projects. To completely remove a project or user from the server use the **Delete Project** and **Delete User** buttons.

Firewall settings

The Unity Asset Server uses TCP port **10733**. You might need to enable connections to this port in your firewall and/or router.

Advanced

The Asset Server is built using a modified version of PostgreSQL. Accessing the SQL database directly requires a bit of technical knowledge about SQL and Unix/Linux command lines. User discretion is advised.

Backing up

We have provided a command line tool to back up an asset server. The tool should be run from an administrator account on the machine running the asset server. Replace BACKUP_LOCATION with the path name you want the backup tool to place the backups:

Mac OS X `sudo /Library/UnityAssetServer/bin/as_backup` BACKUP_LOCATION

Linux `sudo /opt/unity_asset_server/bin/as_backup` BACKUP_LOCATION

Windows `"%ProgramFiles%\Unity\AssetServer\bin\as_backup.cmd"` BACKUP_LOCATION

as_backup will create a directory at BACKUP_LOCATION containing one or more files per project plus files containing information about each project and a backup of all users and their passwords.

Restoring a Backup

To restore an Asset Server backup produced with as_backup, first perform a clean installation of the Asset Server without any projects created. (The restore procedure will refuse to overwrite already existing projects with the same name.)

Then run the provided backup restoration tool, `as_restore` pointing it to the location of a backup created with as_backup:

Mac OS X `sudo /Library/UnityAssetServer/bin/as_restore` BACKUP_LOCATION

Linux `sudo /opt/unity_asset_server/bin/as_restore` BACKUP_LOCATION

Windows `"%ProgramFiles%\Unity\AssetServer\bin\as_restore.cmd"` BACKUP_LOCATION

Note that you can also use as_backup and as_restore to move an asset server installation from one machine to another by performing the backup on the source machine, moving the backup directory to the destination machine (or mount it through a network file share,) and then running as_restore to insert the data into the newly installed Asset Server instance. This will even work when the source and destination Asset Servers have different versions or are running on different operating systems.

Locating the database name of an Asset Server Project

To view the tables in a Project database, first you need to figure out the name of the actual database. Run this command line command on the machine hosting the Asset Server:

Mac OS X `/Library/UnityAssetServer/bin/psql -U admin -h localhost -d postgres -c 'select * from all_databases**view'`

Linux `/opt/unity_asset_server/bin/psql -U admin -h localhost -d postgres -c 'select * from all_databases**view'`

Windows `"%ProgramFiles%\Unity\AssetServer\bin\psql.exe" -U admin -h localhost -d postgres -c "select * from all_databases**view"`

This and other commands will prompt you for a password. Every time this happens, enter the admin password for the database, which was set during the installation. The result will be a table that follows this basic layout:

```
    databasename | projectname | description | version
--------------------+--------------------+--------------------------+---------
 sandbox | Sandbox | Created with Unity 2.0.0 | 1.0
 game | Game | Created with Unity 2.0.0 | 1.0
 my_game_project | My Game Project | Created with Unity 2.0.0 | 1.0
(3 rows)
```

Now you need to identify the “databasename” of the Project you want to back up. When creating a database, the default “databasename” is same as the “projectname” as shown inside Unity, but in lowercase and spaces replaced with underscores.

Note that if your server hosts multiple PostgreSQL databases on different ports you nay need to explicitly provide the port used to connect to the Asset Server database. In this case add -p 10733 to the commands given (assuming you have used the default port of

10733 for your instance.) For example:

Linux `/opt/unity_asset_server/bin/psql -U admin -h localhost -d postgres -c 'select * from all_databases**view' -p 10733`

Additional SQL Functions

These and all other commands use tools from the PostgreSQL distribution. You can read more about these tools here: http://www.postgresql.org/docs/8.3/interactive/reference-client.html

---

2017–06–30 Page amended with no editorial review

Asset Server deprecated in 2017.1

Windows XP support in Standalone Player removed in 2018.1

Legacy Asset Bundles

Leave feedback

Unity 4.x has a different **Asset** Bundle system compared to 5.x. The following pages document the legacy (4.x) asset bundles.

Creating Asset Bundles in Unity 4

Leave feedback

*In versions of Unity earlier than Unity 5, assets had to be selected and added to bundles using editor **scripts** alone. (In Unity 5 we provide tools in the editor to assign assets to specific Bundles). This information is provided for those working on legacy projects in Unity 4, and speaks assuming you are using Unity 4.*

To create an Asset Bundle you need to use the BuildPipeline editor class. All scripts using Editor classes must be placed in a folder named Editor, anywhere in the Assets folder. Here is an example of such a script in C#:

```
// Legacy Unity 4 example. Not for use in Unity 5 & onwards
using UnityEngine;
using UnityEditor;

public class ExportAssetBundles {
    [MenuItem("Assets/Build AssetBundle")]
    static void ExportResource () {
        string path = "Assets/myAssetBundle.unity3d";
        Object[] selection = Selection.GetFiltered(typeof(Object), SelectionMode.DeepAss
        BuildPipeline.BuildAssetBundle(Selection.activeObject, selection, path,
                                       BuildAssetBundleOptions.CollectDependencies
                                     | BuildAssetBundleOptions.CompleteAssets);
    }
}
```

An Editor script does not need to be applied to a **GameObject**, it is instead used by the Editor. This previous example will create a new item in the “Assets” menu of your editor called “Build AssetBundle”.

To use this example:

Create a C# script file named ExportAssetBundles.cs, inside an folder named Editor, in the **Project View**.
Select the Asset or Assets in your project folder that you wish to make into an AssetBundle.
Select Build AssetBundle from the Assets menu. Click “Save” to create the AssetBundle.
The first line of the ExportResource function sets the path of your AssetBundle.
The next line sets the selection to be made into an AssetBundle as the objects selected in the Project window.
The BuildAssetBundle function is the line that creates the AssetBundle and saves it to the specified location. The first parameter specifies the mainAsset, which is a special Asset that can be obtained directly with the mainAsset property when loading Assets from the AssetBundle. It is not mandatory to set a main Asset, if this is not going to be used you can use null for the parameter. The second parameter is the array of objects that will make up the AssetBundle. The third parameter is the location on disk that the AssetBundle will be saved to. The final parameters are the build flags or options used when building AssetBundles. These BuildAssetBundleOptions can be combined using the bitwise OR operator.

Building AssetBundles should be a pre-publish step which happens only once and with a single function call, for example, with a Menu Item that builds all the AssetBundles. As you develop your application you should write helper scripts that can build all your AssetBundles for a target platform with a single click or in batchmode without user intervention.

There are three class methods you can use to build AssetBundles:

BuildPipeline.BuildAssetBundle allows you to build AssetBundles of any type of asset.

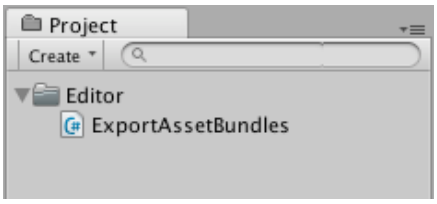
BuildPipeline.BuildStreamedSceneAssetBundle is used when you want to include only **scenes** to be streamed and loaded as the data becomes available.

BuildPipeline.BuildAssetBundleExplicitAssetNames is the same as BuildPipeline.BuildAssetBundle but has an extra parameter to specify a custom string identifier (name) for each object.

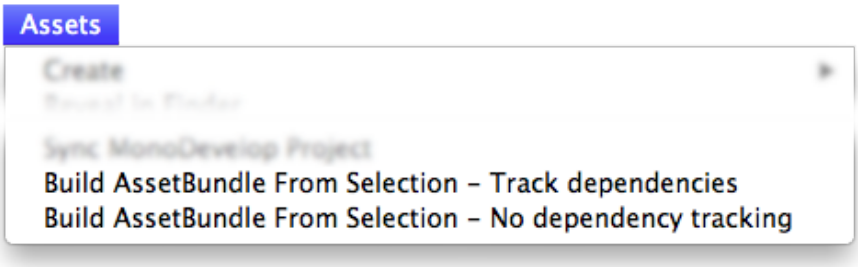
An example of how to build an AssetBundle

Building asset bundles is done through editor scripting. There is basic example of this in the scripting documentation for BuildPipeline.BuildAssetBundle.

For the sake of this example, copy and paste the script from the link above into a new C# script called ExportAssetBundles. This script should be placed in a folder named Editor, so that it works inside the Unity Editor.



Now in the **Assets** menu, you should see two new menu options.

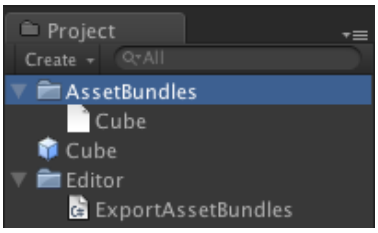


Build AssetBundle From Selection - Track dependencies. This will build the current object into an asset bundle and include all of its dependencies. For example if you have a **prefab** that consists of several hierarchical layers then it will recursively add all the child objects and **components** to the asset bundle.

Build AssetBundle From Selection - No dependency tracking. This is the opposite of the previous method and will only include the single asset you have selected.

For this example, you should create a new prefab. First create a new Cube by going to **GameObject > 3D Object > Cube**, which will create a new cube in the Hierarchy View. Then drag the Cube from the Hierarchy View into the Project View, which will create a prefab of that object.

You should then right click the Cube prefab in the project window and select **Build AssetBundle From Selection - Track dependencies**. At this point you will be presented with a window to save the "bundled" asset. If you created a new folder called "AssetBundles" and saved the cube as **Cube.unity3d**, your project window will now look something like this.



At this point you can move the AssetBundle **Cube.unity3d** elsewhere on your local storage, or upload it to a server of your choice.

An example of how to change the properties of the assets when building an Asset Bundle

You can use AssetDatabase.ImportAsset to force reimporting the asset right before calling BuildPipeline.BuildAssetBundle, and then use AssetPostprocessor.OnPreprocessTexture to set the required properties. The following example will show you how to set different **texture compressions** when building the Asset Bundle.

```
// Legacy Unity 4 example. Not for use in Unity 5 & onwards.
// Builds an asset bundle from the selected objects in the project view,
// and changes the texture format using an AssetPostprocessor.

using UnityEngine;
using UnityEditor;

public class ExportAssetBundles {

    // Store current texture format for the TextureProcessor.
    public static TextureImporterFormat textureFormat;

    [MenuItem("Assets/Build AssetBundle From Selection - PVRTC_RGB2")]
    static void ExportResourceRGB2 () {
        textureFormat = TextureImporterFormat.PVRTC_RGB2;
        ExportResource();
    }

    [MenuItem("Assets/Build AssetBundle From Selection - PVRTC_RGB4")]
    static void ExportResourceRGB4 () {
        textureFormat = TextureImporterFormat.PVRTC_RGB4;
        ExportResource();
    }

    static void ExportResource () {
        // Bring up save panel.
        string path = EditorUtility.SaveFilePanel ("Save Resource", "", "New Resource",

        if (path.Length != 0) {
            // Build the resource file from the active selection.
            Object[] selection = Selection.GetFiltered(typeof(Object), SelectionMode.Dee

            foreach (object asset in selection) {
                string assetPath = AssetDatabase.GetAssetPath((UnityEngine.Object) asset
                if (asset is Texture2D) {
                    // Force reimport thru TextureProcessor.
                    AssetDatabase.ImportAsset(assetPath);
                }
            }

            BuildPipeline.BuildAssetBundle(Selection.activeObject, selection, path, Buil
            Selection.objects = selection;
        }
```

```
    }
}
```

```
// Legacy Unity 4 example. Not for use in Unity 5 & onwards.
// Changes the texture format when building the Asset Bundle.

using UnityEngine;
using UnityEditor;

public class TextureProcessor : AssetPostprocessor
{
    void OnPreprocessTexture() {
        TextureImporter importer = assetImporter as TextureImporter;
        importer.textureFormat = ExportAssetBundles.textureFormat;
    }
}
```

You can also control how the asset is imported using the AssetDatabase.ImportAssetOptions.

In a test environment, you sometimes need to test a change that requires AssetBundles to be rebuilt. In these cases, it is advisable to use the option BuildAssetBundleOptions.UncompressedAssetBundle when you build the AssetBundles. This makes it faster to build and load the AssetBundles but they will also be bigger and therefore take longer to download.

Building AssetBundles in a production enviroment

When first using AssetBundles, it may seem enough to manually build them as seen in the previous example. But as a project grows in size and the number of assets increases, doing this process by hand is not efficient. A better approach is to write a function that builds all of the AssetBundles for a project. You can, for example, use a text file that maps Asset files to AssetBundle files.

Managing Asset Dependencies in Unity 4

Leave feedback

*In versions of Unity earlier than Unity 5, **assets** dependencies had to be defined using editor **scripts** alone. (In Unity 5 we provide tools in the editor to assign assets to specific Bundles and dependency handling is automatic). This information is provided for those working on legacy projects in Unity 4, and speaks assuming you are using Unity 4.*

Any given asset in a bundle may depend on other assets. For example, a model may incorporate materials which in turn make use of textures and **shaders**. It is possible to include all an asset's dependencies along with it in its bundle. However, several assets from different bundles may all depend on a common set of other assets (eg, several different models of buildings may use the same brick texture). If a separate copy of a shared dependency is included in each bundle that has objects using it, then redundant instances of the assets will be created when the bundles are loaded. This will result in wasted memory.

To avoid such wastage, it is possible to separate shared dependencies out into a separate bundle and simply reference them from any bundles with assets that need them. First, the referencing feature needs to be enabled with a call to BuildPipeline.PushAssetDependencies. Then, the bundle containing the referenced dependencies needs to be built. Next, another call to PushAssetDependencies should be made before building the bundles that reference the assets from the first bundle. Additional levels of dependency can be introduced using further calls to PushAssetDependencies. The levels of reference are stored on a stack, so it is possible to go back a level using the corresponding BuildPipeline.PopAssetDependencies function. The push and pop calls need to be balanced including the initial push that happens before building.

At runtime, you need to load a bundle containing dependencies before any other bundle that references them. For example, you would need to load a bundle of shared textures before loading a separate bundle of materials that reference those textures.

Asset IDs

If you anticipate needing to rebuild asset bundles that are part of a dependency chain then you should build them with the BuildAssetBundleOptions.DeterministicAssetBundle option enabled. This guarantees that the internal ID values used to identify assets will be the same each time the bundle is rebuilt.

When building the asset bundle with this method, the objects in it are assigned a 32 bit hash code that is calculated using the name of the asset bundle file, the GUID of the asset and the local id of the object in the asset. For that reason make sure to use the same file name when rebuilding. Also note that having a lot of objects might cause hash **collisions** preventing Unity from building the asset bundle.

Shaders dependencies

Whenever shaders are directly referenced as parameters in BuildPipeline.BuildAssetBundle, or indirectly with the option BuildAssetBundleOptions.CollectDependencies the shader's code is included with the asset bundle. This could cause a problem if you use BuildAssetBundle alone to create several asset bundles, since referenced shaders will be included in every generated bundle. There could be conflicts, i.e. when you mix different versions of a shader, so you will have to rebuild all your bundles after modifying the shaders. The shader's code will also increase the size of bundles. To avoid these problems you can use BuildPipeline.PushAssetDependencies to separate shaders in a single bundle, and that will allow you to update the shader bundle only. As an example of how to achieve this workflow, you can create a **prefab** that includes references to the required shaders:

C# Example

```
using UnityEngine;

public class ShadersList : MonoBehaviour {
    public Shader[] list;
}
```

Create an empty object, assign the script, add the shaders to the list and create the prefab, i.e. “ShadersList”. Then you can create an exporter that generates all the bundles and updates the bundle of shaders:

C# Example

```
using UnityEngine;
using UnityEditor;

public class Exporter : MonoBehaviour {

    [MenuItem("Assets/Export all asset bundles")]
    static void Export() {
        BuildAssetBundleOptions options =
            BuildAssetBundleOptions.CollectDependencies |
            BuildAssetBundleOptions.CompleteAssets |
            BuildAssetBundleOptions.DeterministicAssetBundle;

        BuildPipeline.PushAssetDependencies();
        BuildPipeline.BuildAssetBundle(AssetDatabase.LoadMainAssetAtPath("Assets/S

        BuildPipeline.PushAssetDependencies();
        BuildPipeline.BuildAssetBundle(AssetDatabase.LoadMainAssetAtPath("Assets/S
        BuildPipeline.BuildAssetBundle(AssetDatabase.LoadMainAssetAtPath("Assets/S

        BuildPipeline.PopAssetDependencies();
        BuildPipeline.PopAssetDependencies();
    }

    [MenuItem("Assets/Update shader bundle")]
    static void ExportShaders() {
        BuildAssetBundleOptions options =
            BuildAssetBundleOptions.CollectDependencies |
            BuildAssetBundleOptions.CompleteAssets |
            BuildAssetBundleOptions.DeterministicAssetBundle;
```

```
        BuildPipeline.PushAssetDependencies();
        BuildPipeline.BuildAssetBundle(AssetDatabase.LoadMainAssetAtPath("Assets/S

        BuildPipeline.PopAssetDependencies();
    }
}
```

Bear in mind that you must load the shader bundle first. One drawback of this method is that the option BuildAssetBundleOptions.DeterministicAssetBundle can produce conflicts due to colliding hashes when the amount of objects is too large. In this case the build will fail, and it won't be possible to update the shader bundle alone. In this case you will have to remove that option and rebuild all the asset bundles.

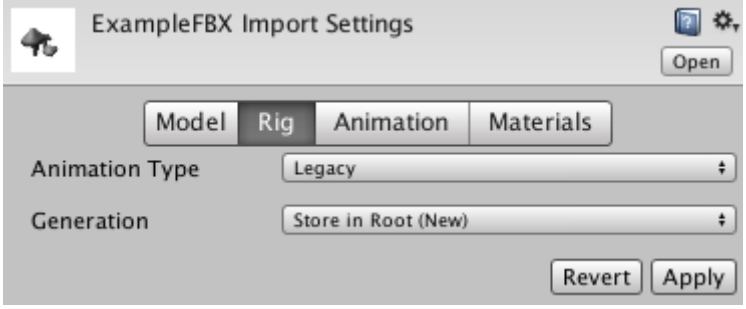
Legacy Animation System

Leave feedback

Prior to the introduction of Mecanim, Unity used a simpler animation system. For backward compatibility, this system is still available. The main reason for using legacy animation is to continue working with an old project without the work of updating it for Mecanim. However, it is not recommended that you use the legacy system for new projects.

Working with legacy animations

To import a legacy animation, you first need to select **Legacy** as the **Animation Type** in the Rig tab of the Import Settings window:



Rig tab on the Import Settings window

Make sure you enable the **Import Animation** option on the Animation tab.

2018–04–25 Page amended with limited editorial review

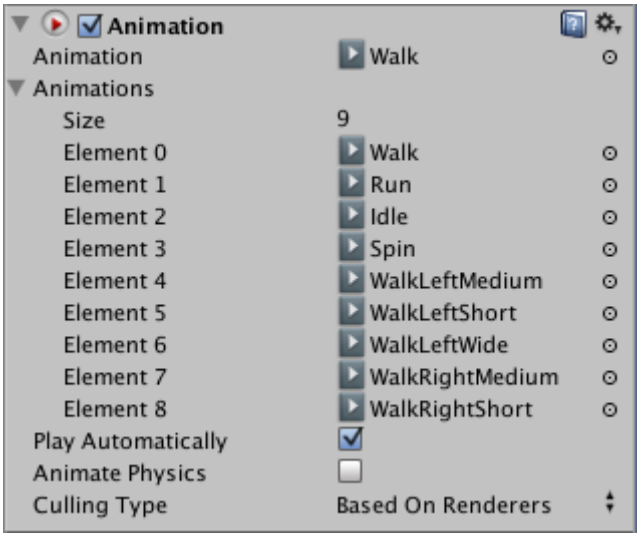
Animation

Leave feedback

SWITCH TO SCRIPTING

This is the **Legacy** Animation **component**, which was used on **GameObjects** for animation purposes prior to the introduction of Unity's current animation system.

This component is retained in Unity for backwards-compatibility only. For new projects, please use the Animator component.



The Animation Inspector

Properties

Property:	*Function:*
Animation	The default animation to play when **Play Automatically** is enabled.
Animations	A list of animations that you can access from **scripts**.
Play Automatically	Enable this option to play animation automatically when the game starts.
Animate Physics	Enable this option to have the animation interact with Physics.
Culling Type	Determine when not to play the animation.
Always Animate	Always animate.
Based on Renderers	Cull based on the default animation pose.
Based on Clip Bounds	Cull based on clip bounds (calculated during import). The animation does not be play when the clip bounds are out of view.
Based on User Bounds	Cull based on bounds defined by the user. The animation does not be play when the user-defined bounds are out of view.

See the Animation Window Guide for more information on how to create animations inside Unity. See Model import workflows page on how to import animated characters.

2018–04–25 Page amended with limited editorial review

Animation Scripting (Legacy)

Leave feedback

Unity's Animation System allows you to create beautifully animated skinned characters. The Animation System supports animation blending, mixing, additive animations, walk cycle time synchronization, **animation layers**, control over all aspects of the animation playback (time, speed, blend-weights), **mesh skinning** with 1, 2 or 4 bones per vertex as well as supporting physically based rag-dolls and procedural animation. To obtain the best results, it is recommended that you read about the best practices and techniques for creating a rigged character with optimal performance in Unity on the Modeling Optimized Characters page.

Making an animated character involves two things; *moving* it through the world and *animating* it accordingly. If you want to learn more about moving characters around, take a look at the Character Controller page. This page focuses on the animation. The actual animating of characters is done through Unity's scripting interface.

You can download example demos showing pre-setup animated characters. Once you have learned the basics on this page you can also see the animation script interface.

Animation Blending

In today's games, animation blending is an essential feature to ensure that characters have smooth animations. Animators create separate animations, for example, a walk cycle, run cycle, idle animation or shoot animation. At any point in time during your game you need to be able to transition from the idle animation into the walk cycle and vice versa. Naturally, you want the transition to be smooth and avoid sudden jerks in the motion.

This is where animation blending comes in. In Unity you can have any number of animations playing on the same character. All animations are blended or added together to generate the final animation.

Our first step will be to make a character blend smoothly between the idle and walk animations. In order to make the scripter's job easier, we will first set the **Wrap Mode** of the animation to **Loop**. Then we will turn off **Play Automatically** to make sure our script is the only one playing animations.

Our first script for animating the character is quite simple; we only need some way to detect how fast our character is moving, and then fade between the walk and idle animations. For this simple test, we will use the standard input axes:-

```
function Update () {
   if (Input.GetAxis("Vertical") > 0.2)
       animation.CrossFade ("walk");
   else
      animation.CrossFade ("idle");
}
```

To use this script in your project:-

Create a Javascript file using **Assets > Create > Javascript**.
Copy and paste the code into it
Drag the script onto the character (it needs to be attached to the **GameObject** that has the animation)
When you hit the Play button, the character will start walking in place when you hold the up arrow key and return to the idle pose when you release it.

Animation Layers

Layers are an incredibly useful concept that allow you to group animations and prioritize weighting.

Unity's animation system can blend between as many **animation clips** as you want. You can assign blend weights manually or simply use **animation.CrossFade()**, which will animate the weight automatically.

Blend weights are always normalized before being applied

Let's say you have a walk cycle and a run cycle, both having a weight of 1 (100%). When Unity generates the final animation, it will normalize the weights, which means the walk cycle will contribute 50% to the animation and the run cycle will also contribute 50%.

However, you will generally want to prioritize which animation receives most weight when there are two animations playing. It is certainly possible to ensure that the weight sums up to 100% manually, but it is easier just to use layers for this purpose.

Layering Example

As an example, you might have a shoot animation, an idle and a walk cycle. The walk and idle animations would be blended based on the player's speed but when the player shoots, you would want to show only the shoot animation. Thus, the shoot animation essentially has a higher priority.

The easiest way to do this is to simply keep playing the walk and idle animations while shooting. To do this, we need to make sure that the shoot animation is in a higher layer than the idle and walk animations, which means the shoot animation will receive blend weights first. The walk and idle animations will receive weights only if the shoot animation doesn't use all 100% of the blend weighting. So, when CrossFading the shoot animation in, the weight will start out at zero and over a short period become 100%. In the beginning the walk and idle layer will still receive blend weights but when the shoot animation is completely faded in, they will receive no weights at all. This is exactly what we need!

```
function Start () {
   // Set all animations to loop
   animation.wrapMode = WrapMode.Loop;
   // except shooting
   animation["shoot"].wrapMode = WrapMode.Once;

   // Put idle and walk into lower layers (The default layer is always 0)
   // This will do two things
   // - Since shoot and idle/walk are in different layers they will not affect
   // each other's playback when calling CrossFade.
   // - Since shoot is in a higher layer, the animation will replace idle/walk
   // animations when faded in.
   animation["shoot"].layer = 1;

   // Stop animations that are already playing
   //(In case user forgot to disable play automatically)
   animation.Stop();
}

function Update () {
   // Based on the key that is pressed,
   // play the walk animation or the idle animation
   if (Mathf.Abs(Input.GetAxis("Vertical")) > 0.1)
      animation.CrossFade("walk");
   else
      animation.CrossFade("idle");
```

```
    // Shoot
    if (Input.GetButtonDown ("Fire1"))
       animation.CrossFade("shoot");
}
```

By default the **animation.Play()** and **animation.CrossFade()** will stop or fade out animations that are in the same layer. This is exactly what we want in most cases. In our shoot, idle, run example, playing idle and run will not affect the shoot animation and vice versa (you can change this behavior with an optional parameter to animation.CrossFade if you like).

Animation Mixing

Animation mixing allow you to cut down on the number of animations you need to create for your game by having some animations apply to part of the body only. This means such animations can be used together with other animations in various combinations.

You add an animation mixing transform to an animation by calling **AddMixingTransform()** on the given AnimationState.

Mixing Example

An example of mixing might be something like a hand-waving animation. You might want to make the hand wave either when the character is idle or when it is walking. Without animation mixing you would have to create separate hand waving animations for the idle and walking states. However, if you add the shoulder transform as a mixing transform to the hand waving animation, the hand waving animation will have full control only from the shoulder **joint** to the hand. Since the rest of the body will not be affected by the hand-waving, it will continue playing the idle or walk animation. Consequently, only the one animation is needed to make the hand wave while the rest of the body is using the idle or walk animation.

```
/// Adds a mixing transform using a Transform variable
var shoulder : Transform;
animation["wave_hand"].AddMixingTransform(shoulder);
```

Another example using a path.

```
function Start () {
   // Adds a mixing transform using a path instead
   var mixTransform : Transform = transform.Find("root/upper_body/left_shoulder");
   animation["wave_hand"].AddMixingTransform(mixTransform);
}
```

Additive Animations

Additive animations and animation mixing allow you to cut down on the number of animations you have to create for your game, and are important for creating facial animations.

Suppose you want to create a character that leans to the sides as it turns while walking and running. This leads to four combinations (walk-lean-left, walk-lean-right, run-lean-left, run-lean-right), each of which needs an animation. Creating a separate animation for each combination clearly leads to a lot of extra work even in this simple case but the number of combinations increases dramatically with each additional action. Fortunately additive animation and mixing avoids the need to produce separate animations for combinations of simple movements.

Additive Animation Example

Additive animations allow you to overlay the effects of one animation on top of any others that may be playing. When generating additive animations, Unity will calculate the difference between the first frame in the animation clip and the current frame. Then it will apply this difference on top of all other playing animations.

Referring to the previous example, you could make animations to lean right and left and Unity would be able to superimpose these on the walk, idle or run cycle. This could be achieved with code like the following:-

```
private var leanLeft : AnimationState;
private var leanRight : AnimationState;

function Start () {
   leanLeft = animation["leanLeft"];
   leanRight = animation["leanRight"];

   // Put the leaning animation in a separate layer
   // So that other calls to CrossFade won't affect it.
   leanLeft.layer = 10;
   leanRight.layer = 10;

   // Set the lean animation to be additive
   leanLeft.blendMode = AnimationBlendMode.Additive;
   leanRight.blendMode = AnimationBlendMode.Additive;

   // Set the lean animation ClampForever
   // With ClampForever animations will not stop
   // automatically when reaching the end of the clip
   leanLeft.wrapMode = WrapMode.ClampForever;
   leanRight.wrapMode = WrapMode.ClampForever;

   // Enable the animation and fade it in completely
   // We don't use animation.Play here because we manually adjust the time
   // in the Update function.
   // Instead we just enable the animation and set it to full weight
   leanRight.enabled = true;
   leanLeft.enabled = true;
   leanRight.weight = 1.0;
   leanLeft.weight = 1.0;

   // For testing just play "walk" animation and loop it
   animation["walk"].wrapMode = WrapMode.Loop;
   animation.Play("walk");
```

```
}

// Every frame just set the normalized time
// based on how much lean we want to apply
function Update () {
   var lean = Input.GetAxis("Horizontal");
   // normalizedTime is 0 at the first frame and 1 at the last frame in the clip
   leanLeft.normalizedTime = -lean;
   leanRight.normalizedTime = lean;
}
```

Tip: When using Additive animations, it is critical that you also play some other non-additive animation on every transform that is also used in the additive animation, otherwise the animations will add on top of the last frame's result. This is most certainly not what you want.

Animating Characters Procedurally

Sometimes you want to animate the bones of your character procedurally. For example, you might want the head of your character to look at a specific point in 3D space which is best handled by a script that tracks the target point. Fortunately, Unity makes this very easy, since bones are just Transforms which drive the skinned mesh. Thus, you can control the bones of a character from a script just like the Transforms of a GameObject.

One important thing to know is that the animation system updates Transforms after the **Update()** function and before the **LateUpdate()** function. Thus if you want to do a **LookAt()** function you should do that in **LateUpdate()** to make sure that you are really overriding the animation.

Ragdolls are created in the same way. You simply have to attach **Rigidbodies**, **Character Joints** and **Capsule Colliders** to the different bones. This will then physically animate your skinned character.

Animation Playback and Sampling

This section explains how animations in Unity are sampled when they are played back by the engine.

AnimationClips are typically authored at a fixed frame rate. For example, you may create your animation in Autodesk® 3ds Max® or Autodesk® Maya® at a frame rate of 60 **frames per second** (fps). When importing the animation in Unity, this frame rate will be read by the importer, so the data of the imported animation is also sampled at 60 **fps**.

However, games typically run at a variable frame rate. The frame rate may be higher on some computers than on others, and it may also vary from one second to the next based on the complexity of the view the **camera** is looking at at any given moment. Basically this means that we can make no assumptions about the exact frame rate the game is running at. What this means is that even if an animation is authored at 60 fps, it may be played back at a different framerate, such as 56.72 fps, or 83.14 fps, or practically any other value.

As a result, Unity must sample an animation at variable framerates, and cannot guarantee the framerate for which it was originally designed. Fortunately, animations for 3D computer graphics do not consist of discrete frames, but rather of continuous curves. These curves can be sampled at any point in time, not just at those points in time that correspond to frames in the original animation. In fact, if the game runs at a higher frame rate than the animation was authored with, the animation will actually look smoother and more fluid in the game than it did in the animation software.

For most practical purposes, you can ignore the fact that Unity samples animations at variable framerates. However, if you have gameplay logic that relies on animations that animate transforms or properties into very specific configurations, then you

need to be aware that the re-sampling takes place behind the **scenes**. For example, if you have an animation that rotates an object from 0 to 180 degrees over 30 frames, and you want to know from your code when it has reached half way there, you should not do it by having a conditional statement in your code that checks if the current rotation is 90 degrees. Because Unity samples the animation according to the variable frame rate of the game, it may sample it when the rotation is just below 90 degrees, and the next time right after it reached 90 degrees. If you need to be notified when a specific point in an animation is reached, you should use an AnimationEvent instead.

Note also that as a consequence of the variable framerate sampling, an animation that is played back using **WrapMode.Once** may not be sampled at the exact time of the last frame. In one frame of the game the animation may be sampled just before the end of the animation, and in the next frame the time can have exceeded the length of the animation, so it is disabled and not sampled further. If you absolutely need the last frame of the animation to be sampled exactly, you should use **WrapMode.ClampForever** which will keep sampling the last frame indefinitely until you stop the animation yourself.

Legacy GUI

Leave feedback

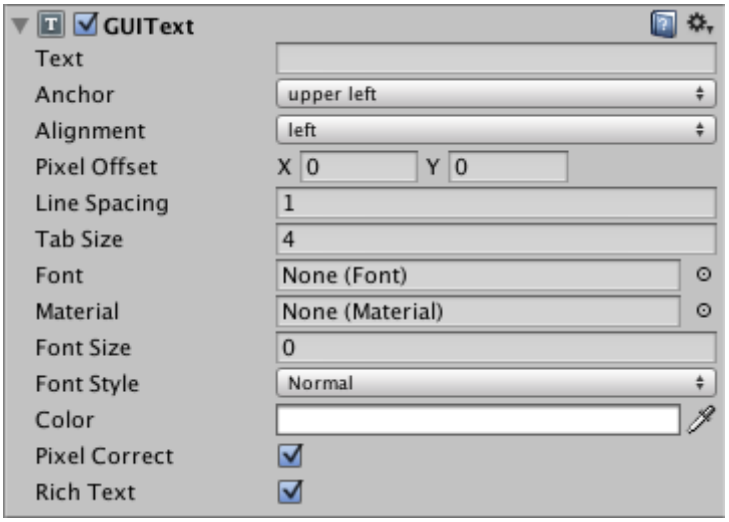
Unity originally had just two GUI **Components**. These have since been replaced by both the modern UI System, and the scripting-driven IMGUI System. The original GUI components are kept for legacy purposes but should not be used for new projects.

GUI Text (Legacy UI Component)

Leave feedback

SWITCH TO SCRIPTING

GUI Text displays text of any font you import in screen coordinates.



Please Note: *This **component** relates to legacy methods for drawing **UI** textures and images to the screen. You should use Unity's up-to-date UI system instead. This is also unrelated to the IMGUI system.*

Properties

Property:	*Function:*
Text	The string to display.
Anchor	The point at which the **Text** shares the position of the **Transform**.
Alignment	How multiple lines are aligned within the **GUIText**.
Pixel Offset	Offset of the text relative to the position of the GUIText in the screen.
Line Spacing	How much space will be in-between lines of **Text**.
Tab Size	How much space will be inserted for a tab ('\t') character. As a multiplum of the space character offset.
Font	The Font to use when **rendering** the text.
Material	Reference to the **Material** containing the characters to be drawn. If set, this property overrides the one in the Font **asset**.
Font Size	The font size to use. Set to 0 to use the default font size. Only applicable for dynamic fonts.
Font Style	The font style to use. (Normal, Bold, Italic or Bold and Italic). Only applicable for dynamic fonts.
Pixel Correct	If enabled, all **Text** characters will be drawn in the size of the imported font texture. If disabled, the characters will be resized based on the Transform's **Scale**.
Rich Text	If enabled, allows HTML-style tags for text formatting.

Details

GUI Texts are used to print text onto the screen in 2D. The **Camera** has to have a GUI Layer attached in order to render the text. Cameras include a GUI Layer by default, so don't remove it if you want to display a GUI Text. GUI Texts are positioned using only the X and Y axes. Rather than being positioned in World Coordinates, GUI Texts are positioned in Screen Coordinates, where (0,0) is the bottom-left and (1,1) is the top-right corner of the screen. To add a GUIText component in Unity 5.0, first use **GameObject->Create Empty** to create an empty game object, then use the **Component->Rendering->GUIText** option to add the GUIText component to the newly created game object. If the text isn't visible when you press Play, check that the transform has suitable position, typically (0.5, 0.5, 0.0).

To import a font see the Font page.

To use Rich Text see the Rich Text page.

Pixel Correct

By default, GUI Texts are rendered with **Pixel Correct** enabled. This makes them look crisp and they will stay the same size in **pixels** independent of the screen resolution.

Hints

When entering text into the **Text** property, you can create a line break by holding **Alt** and pressing **Return**.
If you are scripting the **Text** property, you can add line breaks by inserting the escape character "\n" in your strings.
You can download free true type fonts from 1001freefonts.com (download the Windows fonts since they contain TrueType fonts).

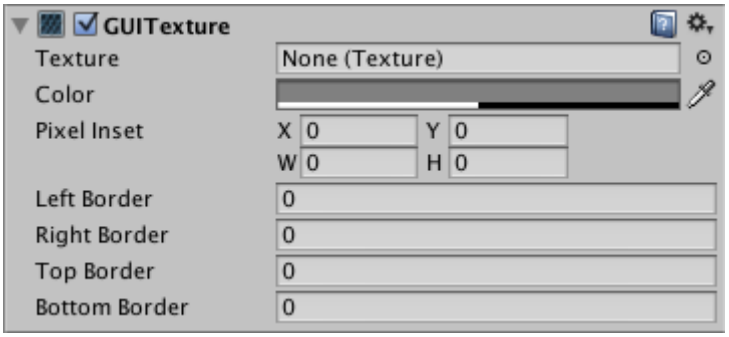
GUI Texture (Legacy UI Component)

Leave feedback

SWITCH TO SCRIPTING

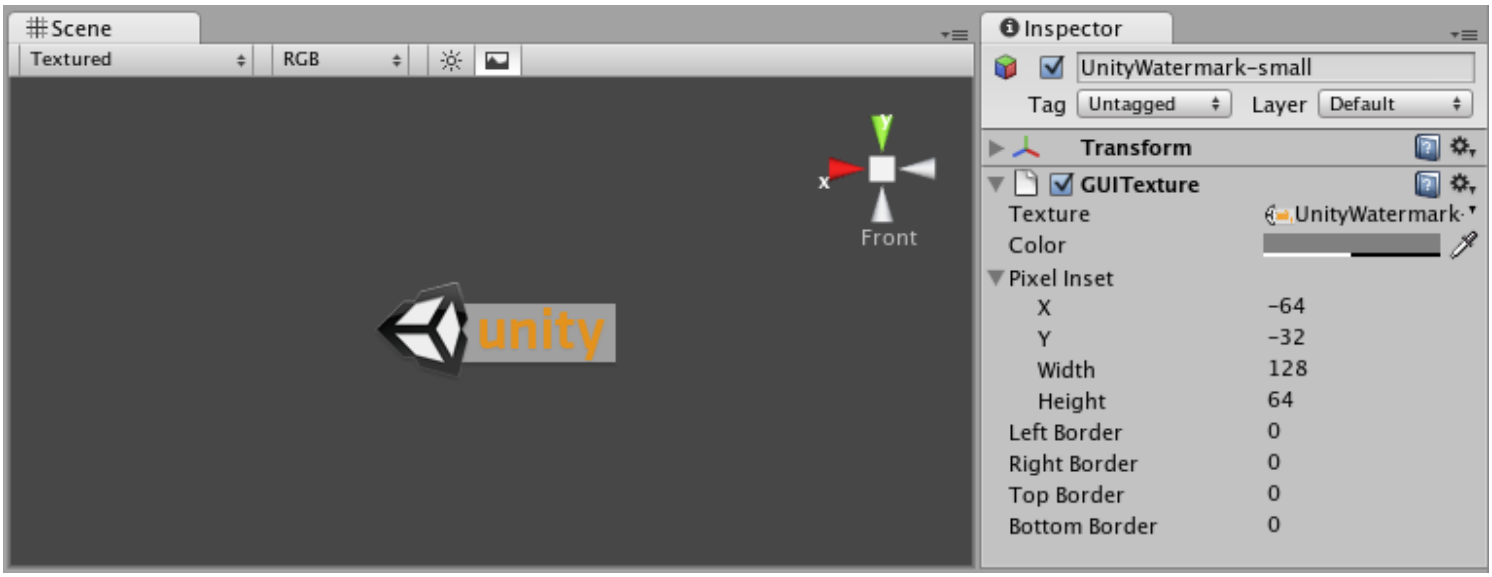
Please Note: *This component relates to legacy methods for drawing UI textures and images to the screen. You should use Unity's up-to-date UI system instead. This is also unrelated to the IMGUI system.*

GUI Textures are displayed as flat images in 2D. They are made especially for user interface elements, buttons, or decorations. Their positioning and scaling is performed along the x and y axes only, and they are measured in **Screen Coordinates**, rather than **World Coordinates**.



Properties

Property:	*Function:*
Texture	Reference to the Texture that will be used as the texture's display.
Color	Color that will tint the **Texture** drawn on screen.
Pixel Inset	Used for pixel-level control of the scaling and positioning of the GUI Texture. All values are measured relative to the position of the GUI Texture's **Transform**.
X	Left-most **pixel** position of the texture.
Y	Bottom-most pixel position of the texture.
Width	Right-most pixel position of the texture.
Height	Top-most pixel position of the texture.
Left Border	Number of pixels from the left that are not affected by scale.
Right Border	Number of pixels from the right that are not affected by scale.
Top Border	Number of pixels from the top that are not affected by scale.
Bottom Border	Number of pixels from the bottom that are not affected by scale.



The GUI Texture Inspector

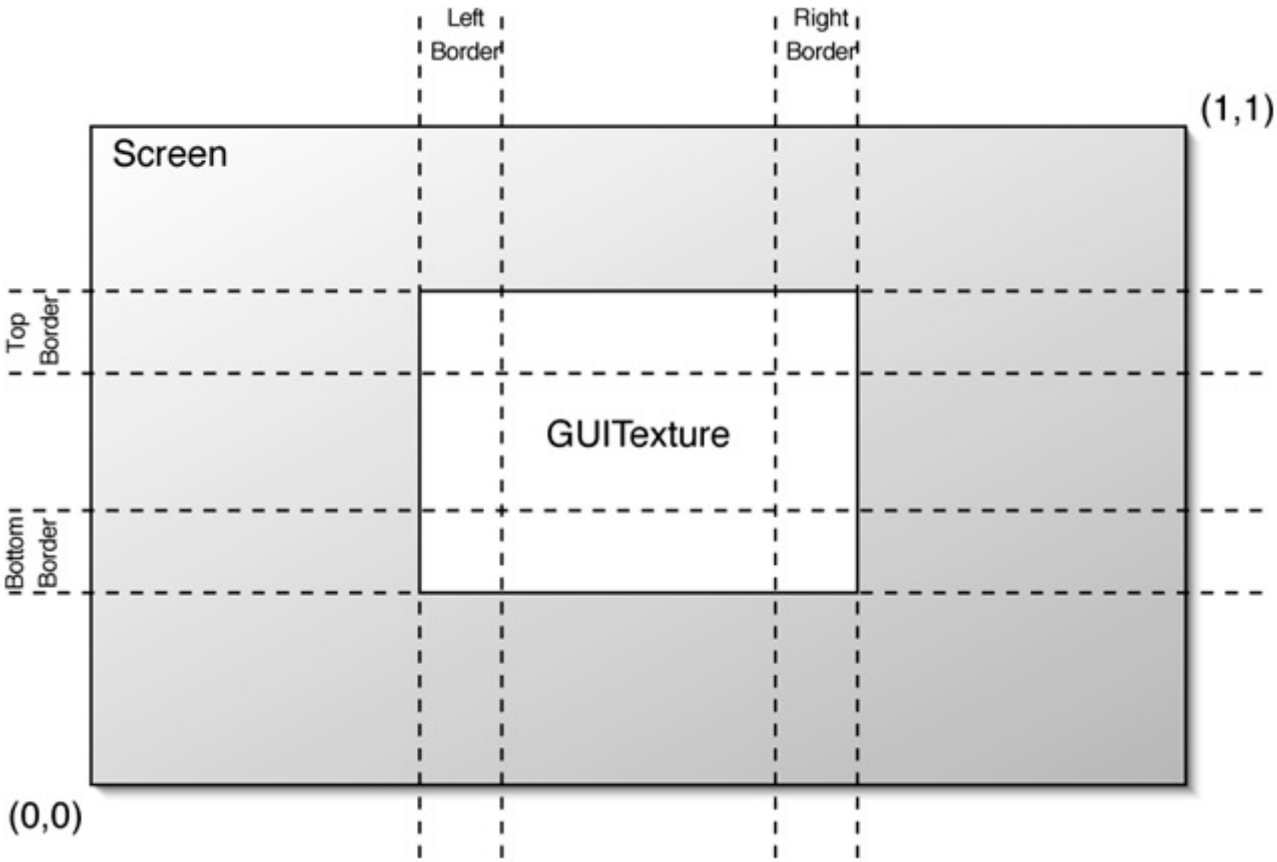
Details

To create a GUITexture:

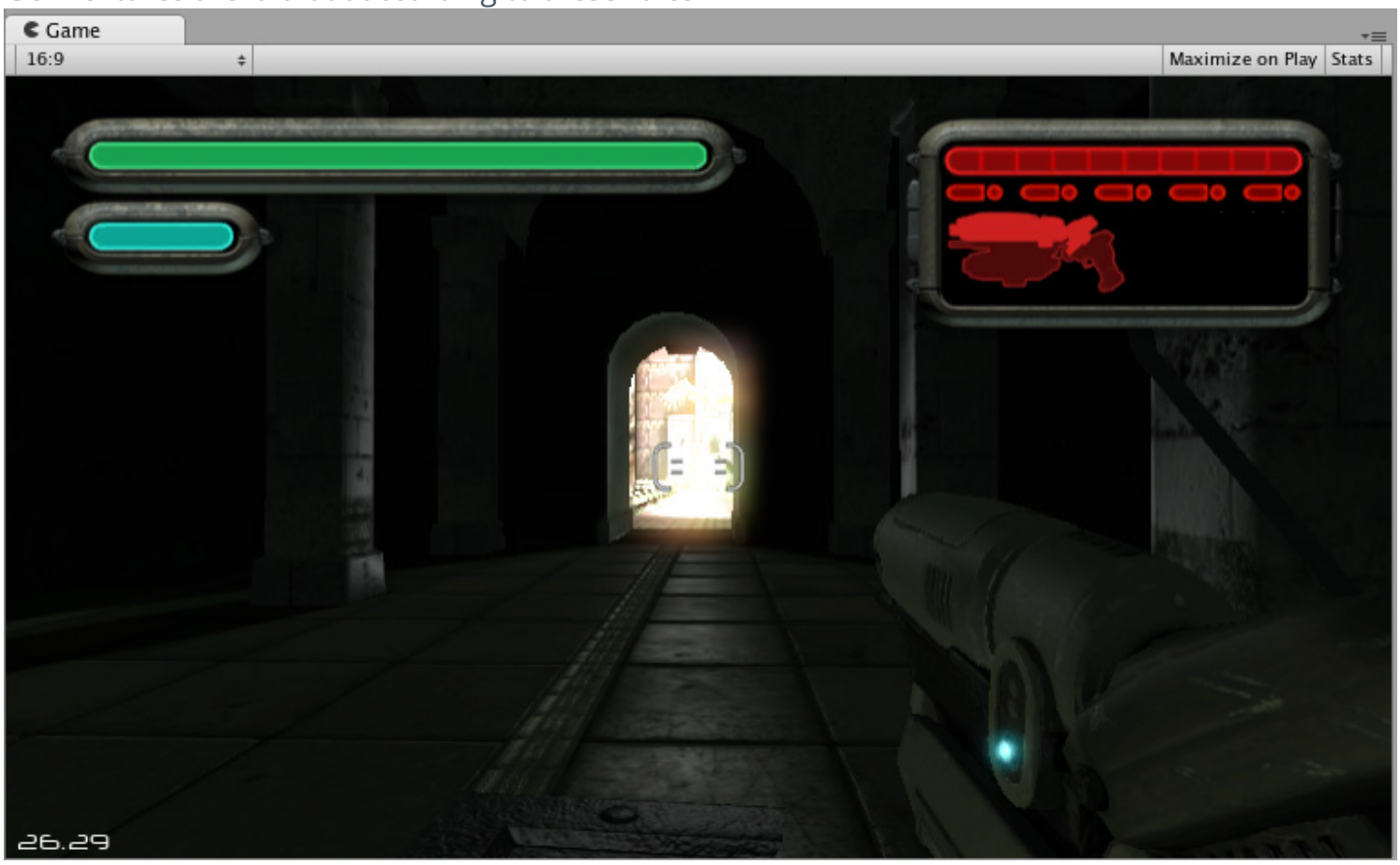
Select a Texture in the **Project View**

Choose **GameObject > Create General > GUI Texture** from the menu bar

GUI Textures are perfect for presenting game interface backgrounds, buttons, or other elements to the player. Through scripting, you can easily provide visual feedback for different "states" of the texture – when the mouse is hovering over the texture, or is actively clicking it for example. Here is the basic breakdown of how the GUI Texture is calculated:



GUI Textures are laid out according to these rules



The GUI elements seen here were all created with GUI Textures

Borders

The borders are the regions at each edge of the image (specified by pixel widths) that will not scale when the rest of the texture is scaled. As you rarely know the resolution your game runs in, chances are your GUI will get scaled. Some GUI textures have a border at the edge that is meant to be an exact number of pixels. In order for this to work, set the border sizes to match those from the texture.

Pixel Inset

The purpose of the **Pixel Inset** is to prevent textures from scaling with screen resolution, and keeping them in a fixed pixel size. This allows you to render a texture without any scaling. This means that players who run your game in higher resolutions will see your textures in smaller areas of the screen, allowing them to have more screen real-estate for your gameplay graphics.

To use it effectively, you need to set the scale of the GUI Texture's Transform to (0, 0, 0). Now, the **Pixel Inset** is in full control of the texture's size and you can set the **Pixel Inset** values to be the exact pixel size of your Texture.

Hints

The depth of each layered GUI Texture is determined by its individual Z Transform position, not the global Z position.
GUI Textures are great for making menu screens, or pause/escape menu screens.
You should use **Pixel Inset** on any GUI Textures that you want to be a specific number of pixels for the width and height.

Legacy Unity Analytics (SDK Workflow)

Leave feedback

Prior to Unity 5.2, Unity **Analytics** support was provided via an SDK which needed to be imported into your project. For new projects, you should use the new built-in Analytics.

These instructions cover using **Unity Analytics** with Unity versions prior to 5.2, and are only recommended for people who are maintaining legacy projects that cannot be upgraded to the latest version of Unity.

To upgrade to 5.2 and up from previous versions see the Unity Analytics Upgrade Instructions.

Basic Integration (SDK)

Leave feedback

Basic Integration has never been quicker and easier. Connecting your unique Unity Project ID to your game triggers Unity **Analytics** to automatically track user engagement and behavioral data for the entire duration of the user's play session.

Import SDK

Leave feedback

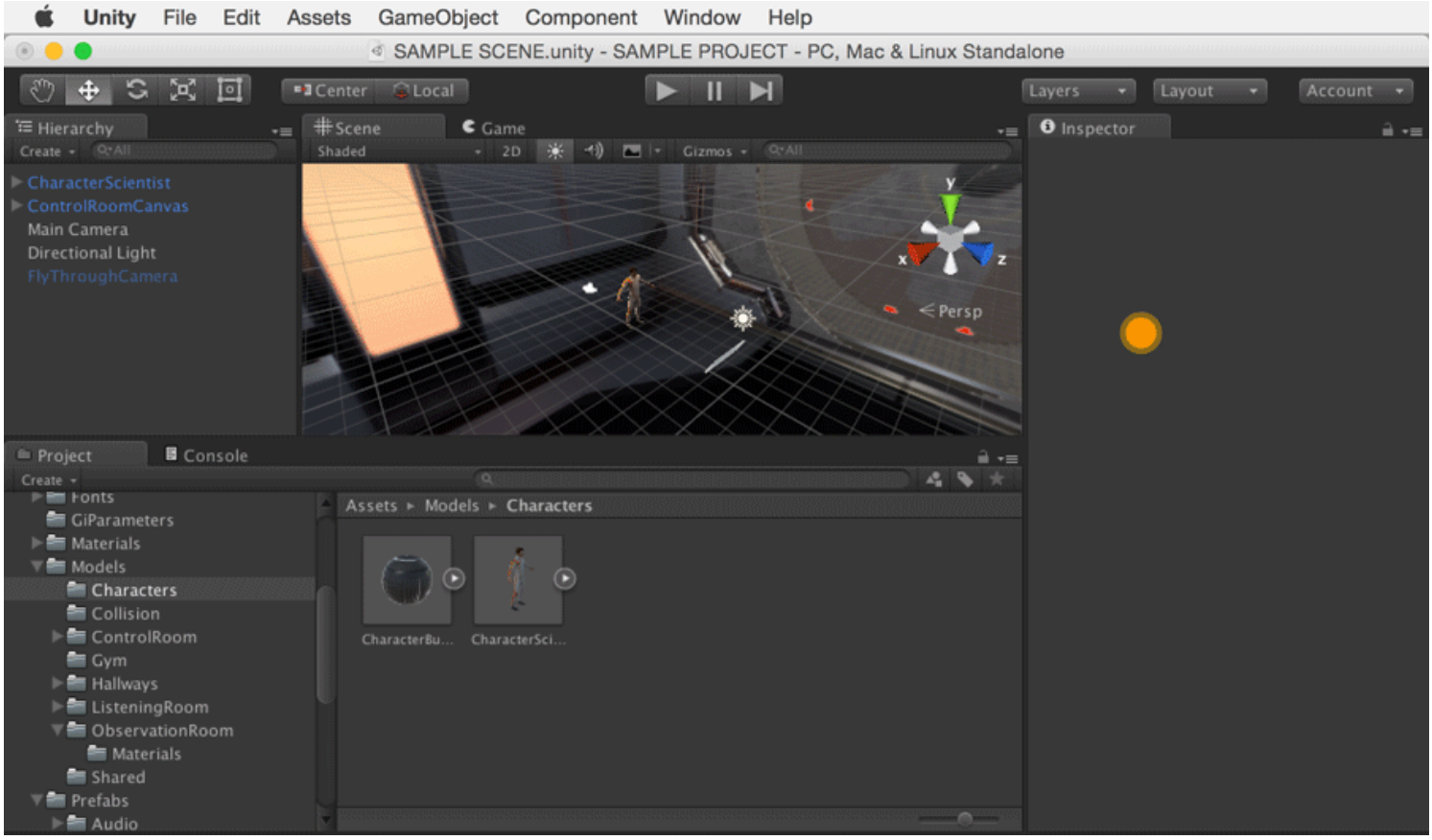
Download Unity Analytics SDK

Download the latest version of the **Unity Analytics** SDK as a Unity package from the Unity Asset Store or download here:

Download from the Analytics Dashboard

Import SDK

To import the downloaded package, go to the Unity Editor and select:
Assets->Import Package->Custom Package
Locate the downloaded SDK package. With the SDK package selected, click “Open”. Leave all the files checked and click **Import**.



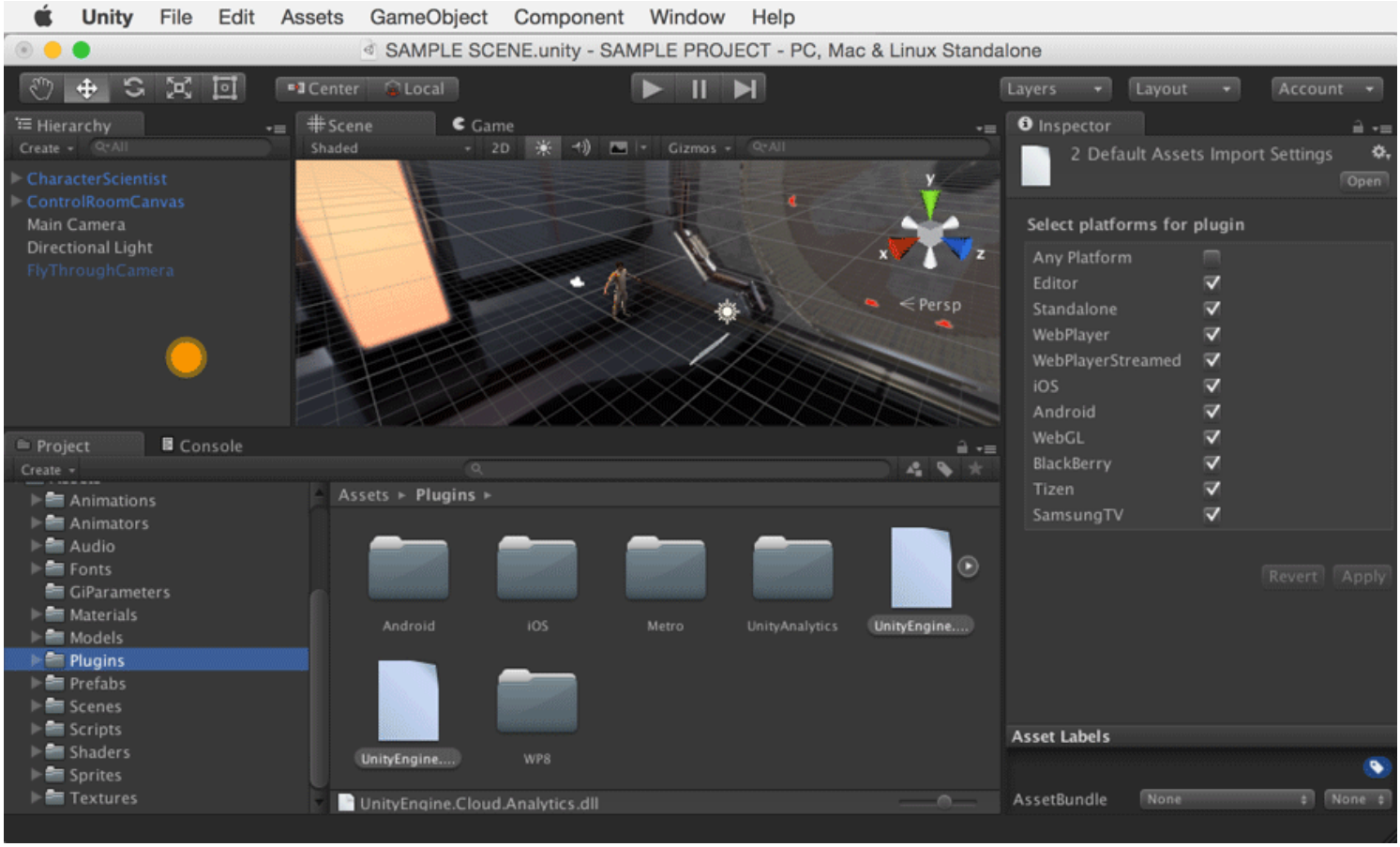
If the import is successful, you should see the Unity Analytics **assets** in the **Project view** panel.

Create Game Script

Leave feedback

Create Script to Start a Player Session

In the Unity Editor, create a new C# script by going to:
Assets->Create->C# Script
Name the newly created script: **UnityAnalyticsIntegration** (naming must be exact for the script to work correctly)



Open Script and Copy Code

Double-click on the script to open your script in your external script editor. Completely replace the default generated code with the code below and save the file in the script editor. The Unity Project ID that you see in the sample code below is unique to your game. It is used to link your Editor project to your **Analytics** dashboard.

Go to the Analytics Dashboard to get your unique, functional Project ID

```
using UnityEngine;
using System.Collections;
using UnityEngine.Cloud.Analytics;

public class UnityAnalyticsIntegration : MonoBehaviour {

    // Use this for initialization
    void Start () {

        const string projectId = "SAMPLE-UNITY-PROJECT-ID";
```

```
        UnityAnalytics.StartSDK (projectId);

    }

}
```

2018–03–19 Page amended with limited editorial review

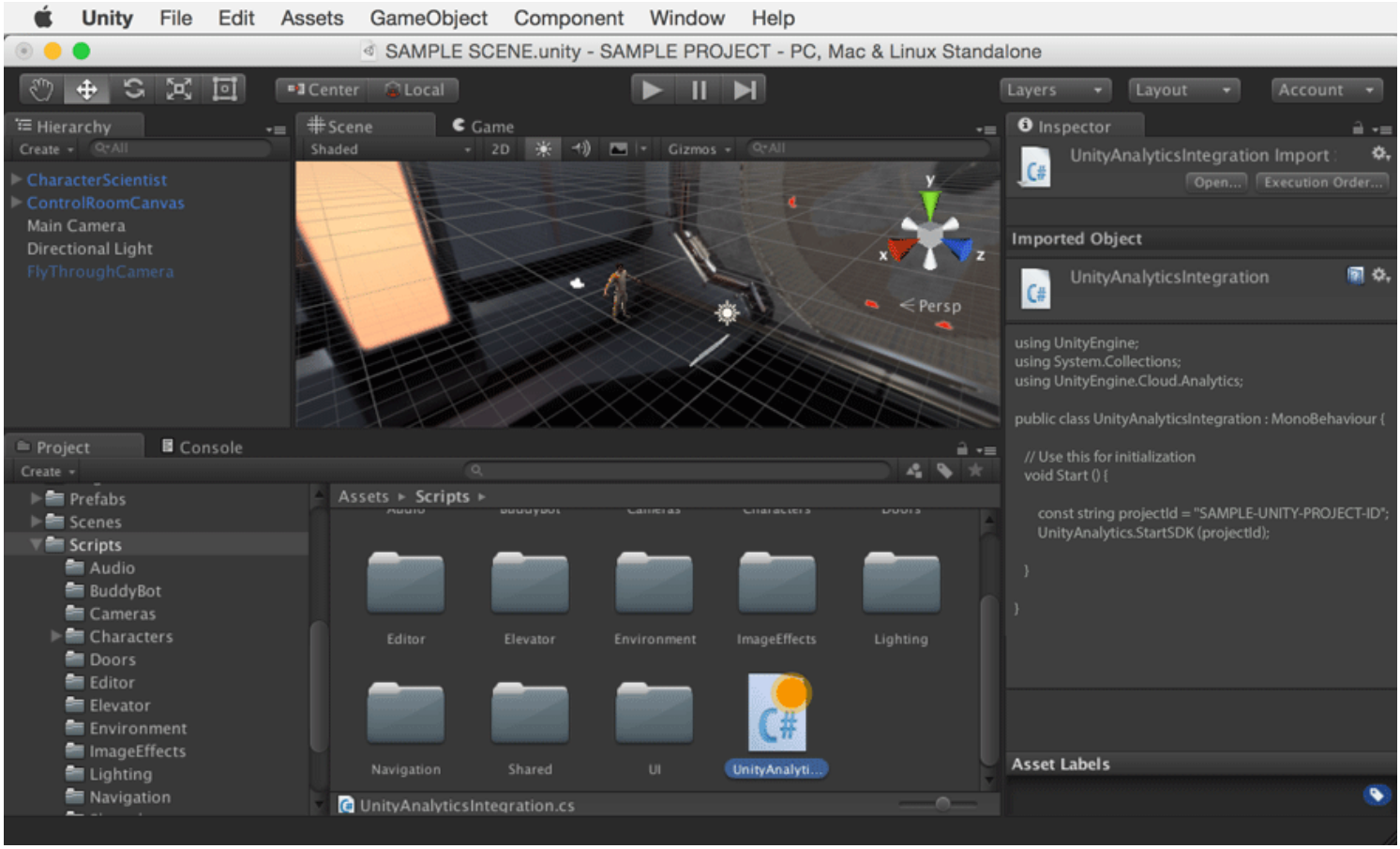
MonoDevelop replaced by Visual Studio from 2018.1

Attach Game Script

Leave feedback

The **UnityAnalyticsIntegration** script can be attached to any GameObject in any Scene in your game to initialize the Unity **Analytics** SDK. We recommend you attach the script to a **GameObject** in the first **Scene** of your game to ensure you begin capturing as much user engagement data as possible.

In this example, we will attach the script to **MainCamera**. First, locate and select "Main **Camera**" in the **Hierarchy** tab on the left hand side of the Editor.



Drag your script from the Projects tab and drop it on the Main Camera **GameObject**. In the **Inspector** window, with Main Camera selected, you should see that the script was assigned to the Main Camera as a **Component**.

Note: You can also add a script to a **GameObject** by selecting the **GameObject** in your **Hierarchy** and adding the desired script Component to it in the **Inspector**.

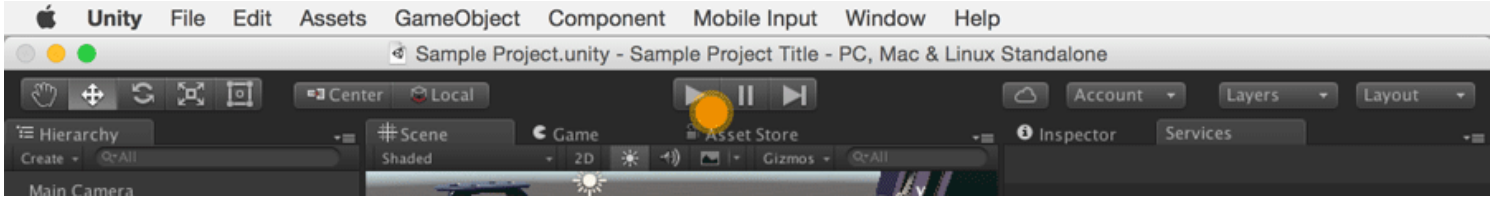
Play To Validate

Leave feedback

The Unity Editor can act as a test environment to validate **Analytics** integration. When you press the Play button, the Editor sends data which you can see in the validator below. This means that it is not required that you build and publish your game in order to validate Basic Integration.

Press Play

To send test data to our servers and validate your integration, simply play your game by pressing the Play button.



Advanced Integration (SDK)

Leave feedback

Advanced Integration provides richer game data and enhances what you can see and do in the dashboard. It is not necessary in order to see basic user engagement **analytics** on the dashboard. Custom Events and Monetization are the building blocks for deeper analysis.

Custom Events

Leave feedback

Custom events can be any specific in-game action your user performs. They allow you to track player behavior Unity **Analytics** does not track automatically, such as level achievement, **scene** change, entering a store, or interacting with game objects. Each custom event can have its own parameters. Setting parameters on your event allows you to filter data collected at the time the event occurred. Visualization tools for Custom Events can be viewed in the **Analytics** Dashboard, including **Data Explorer**, Funnel Analyzer, and Segment Builder.

```
// Reference the Unity Analytics SDK namespace
using UnityEngine.Cloud.Analytics;

//  Use this call for wherever a player triggers a custom event
UnityAnalytics.CustomEvent(string customEventName,
IDictionary<string, object> eventData);
```

UnityAnalytics.CustomEvent Input Parameters

Name	*Type*	*Description*
customEventName	string	Name of custom event. Name cannot include the prefix “unity.” — This is a reserved keyword.
eventData	dictionary	Additional parameters sent to **Unity Analytics** at the time the custom event was triggered. eventData key cannot include the prefix “unity.” — This is a reserved keyword.

A few considerations with regards to the custom events:

Be consistent! Maintain consistent data types for each parameter in your event data. For example, don’t send a level parameter as a number, then change it to be a string. Doing so can lead to erroneous behavior, making your data difficult to interpret.
Boolean (true/false)
String (characters)
Numbers (int, float, etc.).
Default limit of 10 parameters per custom event.
If there are more parameters passed, the call will fail with a return value of AnalyticsResult.TooManyItems
Default limit of 500 characters for the dictionary content.
If more than 500 characters are passed, the call will fail with return value of AnalyticsResult.SizeLimitReached
Default limit of 100 custom events per hour, per user.
If more than 100 events per hour are called, the call will fail with return value of AnalyticsResult.TooManyRequests
Consider how parameters are parsed by the Analytics system.
All numbers, ints, floats, **etc**., even if sent as strings, are parsed as numbers.
Only strings and Booleans are considered ‘categorizable’.

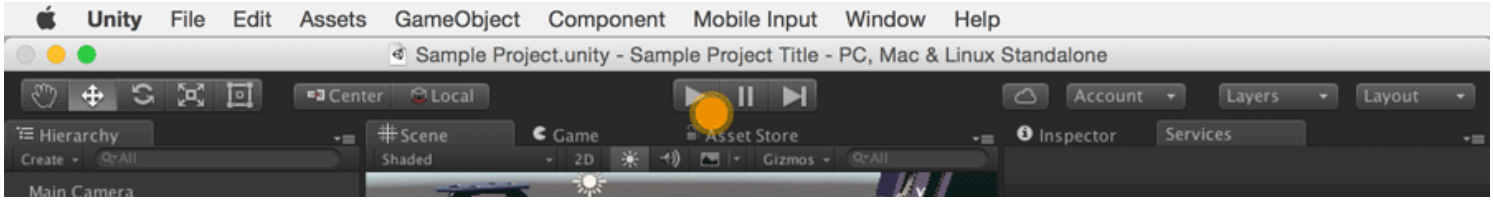
Consequently, if you want something to be summed or averaged, send it as a number (e.g., 51 or '51'). If you want it to be categorized, as you would with a level or option, make sure it will parse as a string (e.g., 'Level51').

In the example below we are interested in knowing what our user had in their inventory at the time the game ended.

```
// Reference the Collections Generic namespace
  using System.Collections.Generic;

  int totalPotions = 5;
  int totalCoins = 100;
  string weaponID = "Weapon_102";
  UnityAnalytics.CustomEvent("gameOver", new Dictionary<string, object>
  {
    { "potions", totalPotions },
    { "coins", totalCoins },
    { "activeWeapon", weaponID }
  });
```

Press Play

To send test Custom Event data to our servers and validate your integration, trigger your Custom Event during Editor Play mode.



Monetization

Leave feedback

Unity **Analytics** allows you to monitor your in-game revenue. By implementing receipt verification you'll quickly see legitimate or fraudulent transactions.

Unity Analytics provides a flexible method for tracking monetization events through in-app purchases. This method should be called every time a player triggers a monetization event. The Transaction method requires a price parameter, a currency and an optional Apple iTunes / Google Play receipt string.

Unity Analytics provides the **UnityAnalytics.Transaction** method for tracking monetization events through in-app purchases. This method should be called every time a player triggers a monetization event. The **UnityAnalytics.Transaction** method requires a price parameter, a currency and an optional Apple iTunes / Google Play receipt string.

```
// Reference the Unity Analytics SDK package
using UnityEngine.Cloud.Analytics;

// Use this call for each and every place that a player triggers a monetizatio
UnityAnalytics.Transaction(string productId, decimal price,
string currency, string receipt,
string signature);
```

UnityAnalytics.Transaction Input Parameters

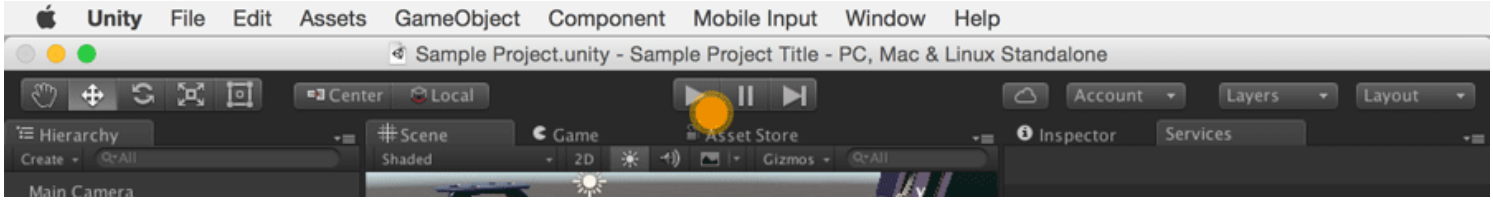
Name	*Type*	*Description*
productId	string	The id of the purchased item.
price	decimal	The price of the item.
currency	string	Abbreviation of the currency used for the transaction. For example "USD" (United States Dollars). See here for a standardized list of currency abbreviations.
receipt	string	Receipt data (iOS) or receipt ID (Android) for in-app purchases to verify purchases with Apple iTunes or Google play. Use **null** in the absence of receipts. For more details see Receipt Verification.
signature	string	Android receipt signature. If using native Android use the **INAPP_DATA_SIGNATURE** string containing the signature of the purchase data that was signed with the private key of the developer. The data signature uses the RSASSA-PKCS1-v1_5 scheme. Pass in **null** in the absence of a signature.

The example below is for a $0.99 transaction in USD without receipt validation.

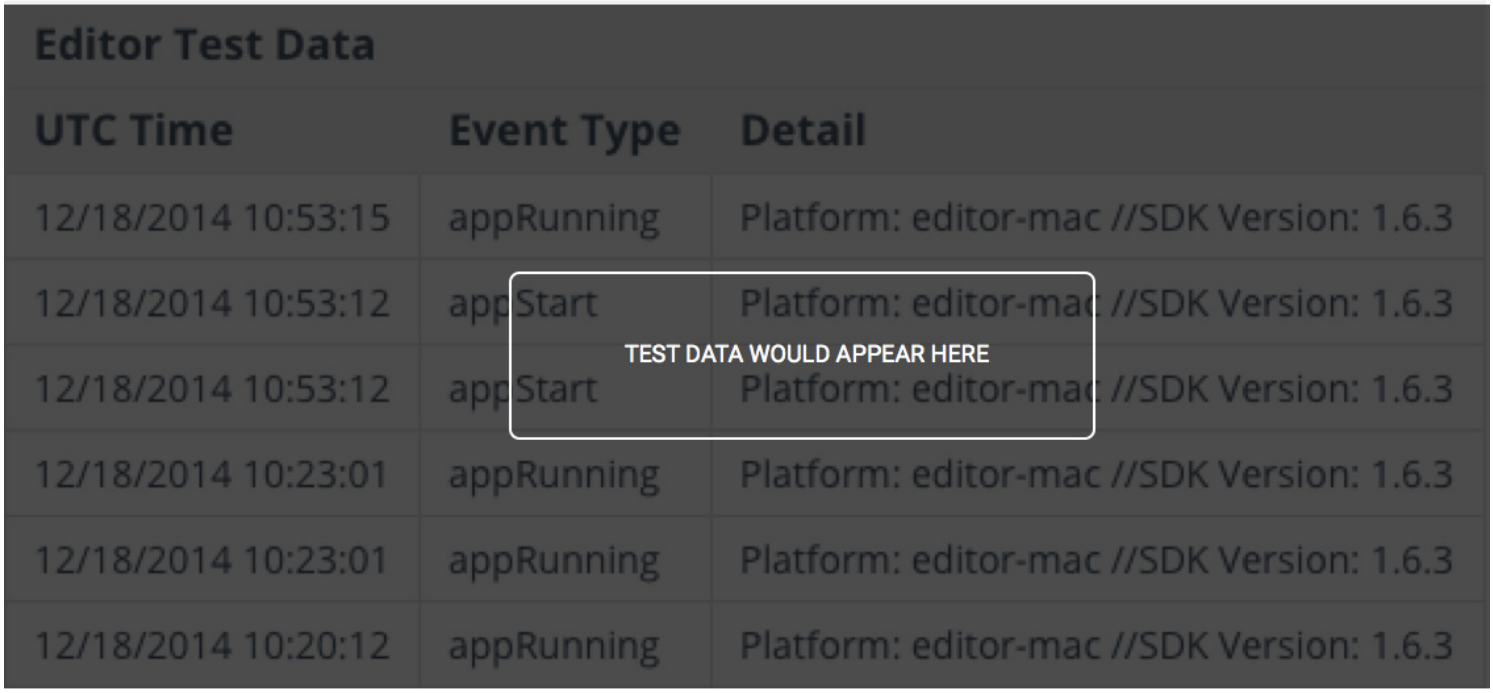
```
UnityAnalytics.Transaction("12345abcde", 0.99m, "USD", null, null);
```

Press Play

To send test Monetization data to our servers and validate your integration, trigger a purchase during Editor Play mode. If integration is successful, your test data will display in the table below.



Validate



Receipt Verification

Leave feedback

Here are the implementation details:

For iOS

receipt parameter

If you leave this as **null**, this transaction will show up in Unverified Revenue
If you are writing a Native **iOS** plugin
For iOS 6 and below, send the raw SKPaymentTransaction's transactionReceipt
For iOS 7 and above, send the base64 encoded value of the App Receipt from the app bundle.
If you are using Unibill plugin
Pass in PurchasableItem's "receipt" property as the receipt.
If you are using Prime31 plugin
Pass in StoreKitTransaction's "base64EncodedTransactionReceipt" property as the receipt.

signature parameter

Pass in **null** since this is not used.

For Android

To validate Android monetization, please enter your Google Public API Key in your **Analytics** dashboard **Project Settings** Form.

The Google Public API Key is needed for implementing receipt verification for in-app purchases (IAP) on Google Play. Your Google Public Key is under Google Play Developer console > All applications > Services & APIs > Your License Key For This Application. This is optional, but if you are developing for Android and have in-app purchases, we highly recommend implementing it.

receipt parameter

If you leave this as null, this transaction will show up as Unverified Revenue
If you are writing a Native Android plugin
Pass in the the purchase data for the order, which is a String in JSON format that is mapped to the INAPP_PURCHASE_DATA key in the response Intent.
If you are using Unibill plugin
Parse the PurchasableItem's "receipt" property as JSON, and pass in the value of the "json" property.
If you are using Prime31 plugin
Pass in GooglePurchase "originalJson" property

signature parameter

If you leave this as **null**, this transaction will show up as Unverified Revenue
If you are writing a Native Android plugin
Pass in the the signature, which is mapped to the INAPP_DATA_SIGNATURE key in the response Intent.
If you are using Unibill plugin
Parse the PurchasableItem's "receipt" property as JSON, and pass in the value of the signature property.
If you are using Prime31 plugin

Pass in GooglePurchase “signature” property.

User Attributes

Leave feedback

Unity **Analytics** offers the ability to track user demographics. This provides you with robust ways to filter your data to look at different user segments. This information is pulled from player information entered at signup or from third-party SDKs.

```
// Reference the Unity Analytics SDK package
  using UnityEngine.Cloud.Analytics;

  // Use this call to designate the user gender
  UnityAnalytics.SetUserGender(Gender gender);

  // Use this call to designate the user birth year
  UnityAnalytics.SetUserBirthYear(int birthYear);
```

Input Parameters

Name	*Type*	*Description*
gender	enum	Gender of user can be Gender.Female, Gender.Male or Gender.Unknown.
birthYear	int	Birth year of user. Must be 4-digit year format only.

For example:

```
  Gender gender = Gender.Female;
  UnityAnalytics.SetUserGender(gender);

  int birthYear = 2014;
  UnityAnalytics.SetUserBirthYear(birthYear);
```

LegacyUnityRemote

Leave feedback

Here you can find older versions of **Unity Remote** for use in Legacy projects.

Unity Remote 3

Unity Remote 3 (DEPRECATED)

Leave feedback

Unity Remote 3 (further refered to as Unity Remote) is an application that allows you to use your **iOS** device as a remote control for your project in Unity. This is useful during development since it is much quicker to test your project in the editor with remote control than to build and deploy it to the device after each change.

DEPRECATED

This is an old page, relating to an old version of **Unity Remote**. Please use the current version of Unity Remote.

Where can I find Unity Remote?

Unity remote is available for download from the AppStore at no charge. If you prefer to build and deploy the application yourself, you can download the source from the Unity Learn section at the Unity website.

How do I build Unity Remote?

First, download the project source code here and unzip it to your preferred location. The zip file contains an XCode project to build Unity Remote and install it on your device.

Assuming you have already created the provisioning profile and successfully installed iOS builds on your device, you just need to open the Xcode project file UnityRemote.xcodeproj. Once XCode is launched, you should click “Build and Go” to install the app on your iOS device. If you have never built and run applications before, we recommend that you try building some of the Apple examples first to familiarize yourself with XCode and iOS.

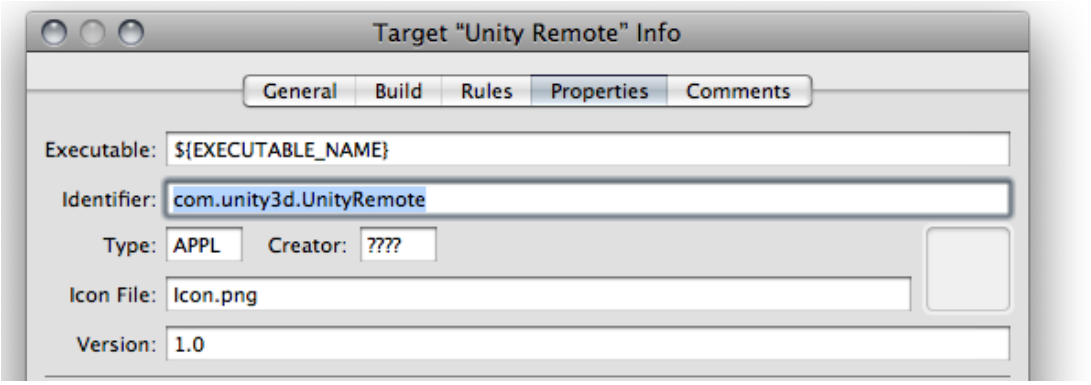
Once Unity Remote is installed, make sure your device is connected via Wi-Fi to the same network as your development machine or else connected to the machine directly via USB. Launch Unity Remote on your iPhone/iPad while Unity is running on your computer and select your computer from the list that appears. Now, whenever you enter Play mode in the Editor, your device will act as a remote control that you can use for developing and testing your game. You can control the application with the device wirelessly and you will also see a low-res version of the app on the device’s screen.

Note: The Unity iOS editor cannot emulate the device’s hardware perfectly, so you may not get the exact behavior (graphics performance, touch responsiveness, sounds playback, etc) that you would on a real device.

Xcode shows strange errors while deploying Unity Remote to my device. What should I do?

This indicates that the default Identifier in the Unity Remote project is not compatible with your provisioning profile. You will have to alter this Identifier manually in your XCode project. The Identifier must match your provisioning profile.

You will need to create an AppID with an trailing asterisk if you have not already done so; you can do this in the Program Portal on Apple’s iPhone Developer Program. First, go to the Program Portal and choose the AppIDs tab. Then, click the Add ID button in the top right corner and type your usual bundle identifier followed by dot and asterisk (eg, com.mycompany.*) in the App ID Bundle Seed ID and Bundle Identifier field. Add the new AppID to your provisioning profile, then download and reinstall it. Don’t forget to restart Xcode afterwards. If you have any problems creating the AppID, consult the Provisioning How-to section on Apple’s website.



Don't forget to change the Identifier before you install Unity Remote on your device.
Open the Unity Remote project with XCode. From the menu, select **Project->Edit Active Target "Unity Remote"**. This will open a new window entitled Target "Unity Remote" Info. Select the Properties tab. Change the Identifier property field from **com.unity3d.UnityRemote** to the bundle identifier in your AppID followed by "**.**" (dot) followed by "**UnityRemote**". For example, if your provisioning profile contains ######.com.mycompany.* AppID, then change the Identifier field to **com.mycompany.UnityRemote**.

Next, select **Build->Clean all targets** from the menu, and compile and install Unity Remote again. You may also need to change the active SDK from Simulator to Device - 2.0 | Release. There is no problem using SDK 2.0 even if your device runs a newer version of the OS.

I'm getting really poor graphics quality when running my game in Unity Remote. What can I do to improve it?

When you use Unity Remote, the game actually runs on your Mac while its visual content is heavily compressed and streamed to the device. As a result, what you see on the device screen is just a low-res version of what the app would really look like. You should check how the game runs on the device occasionally by building and deploying the app (select **File->Build & Run** in the Unity editor).

Unity Remote is laggy. Can I improve it?

The performance of Unity Remote depends heavily on the speed of the Wi-Fi network, the quality of the networking hardware and other factors. For the best experience, create an ad-hoc network between your Mac and iOS device. Click the Airport icon on your Mac and choose "Create Network". Then, enter a name and password and click OK. On the device, choose *Settings->Wi-Fi* and select the new Wi-Fi network you have just created. Remember that an ad-hoc network is really a wireless connection that does not involve a wireless access point. Therefore, you will usually not have internet access while using ad-hoc networking.

Turning Bluetooth off on both on your iPhone/iPad and on Mac should also improve connection quality.

If you do not need to see the game view on the device, you can turn image synchronization off in the Remote machine list. This will reduce the network traffic needed for the Remote to work.

The connection to Unity Remote is easily lost

This can be due to a problem with the installation or other factors that prevent Unity Remote from functioning properly. Try the following steps in sequence, checking if the performance improves at each step before moving on to the next:-

First of all, check if Bluetooth is switched on. Both your Mac and iOS device should have Bluetooth disabled for best performance.
Delete the settings file located at `~/Library/Preferences/com.unity3d.UnityEditoriPhone.plist`
Reinstall the game on your iPhone/iPad.
Reinstall Unity on your Mac.

As a last resort, performing a hard reset on the iOS device can sometimes improve the performance of Unity Remote.

If you still experience problems then try installing Unity Remote on another device (in another location if possible) and see if it gives you better results. There could be problems with RF interference or other software influencing the performance of the wireless adapter on your Mac or iOS device.

Unity Remote doesn't see my Mac. What should I do?

Check if Unity Remote and your Mac are connected to the same wireless network.
Check your firewall settings, router security settings, and any other hardware/software that may filter packets on your network.
Leave Unity Remote running, switch off your Mac's Airport for a minute or two, and switch on again.
Restart both Unity and Unity Remote. Sometimes you also need to cold-restart your iPhone/iPad (hold down the menu and power buttons simultaneously).
Unity Remote uses the Apple Bonjour service, so check that your Mac has it switched on.
Reinstall Unity Remote from the latest Unity iOS package.

Best practice guides

Leave feedback

Learn production-tested best practices from our Enterprise Support engineers and the Unity R&D Content team.

Optimization and visuals

Understanding optimization in Unity
Making believable visuals in Unity
Note that the following links (below) lead to guides on the Unity Learn Website.

Assets, resources and AssetBundles

A guide to AssetBundles and Resources

Assets, objects and serialization

The Resources folder

AssetBundle fundamentals

AssetBundle usage patterns

Optimizing Unity UI

A guide to optimizing Unity UI

Fundamentals of Unity UI

Unity UI profiling tools

Fill-rate, canvases and input

Optimizing UI controls

Other UI optimization techniques and tips

Optimizing Graphics in Unity

Rendering
Lighting Scenes
Auto Generate and Generate Lighting
Lighting Settings
Multi-Scene editing
Camera
Textures
Multithreaded Rendering & Graphics Jobs
Framebuffer
Shaders

Memory Management in Unity

A guide to optimizing Memory
Managed Memory
IL2CPP & Mono
Native Memory
Android Memory Management
Application Size

Profiling in Unity

Profiling applications made with Unity
Unity Profiler
Third Party Profiler
Android Profiling Tools
Intel Tools

Understanding optimization in Unity

Leave feedback

This Best Practice Guide is a companion piece to the Unite Unity Europe 2016 talk *Optimizing Mobile Applications*. It covers much of the same material, but with supplementary material added for the interested reader. You can watch the talk itself on YouTube: Unite 2016: Optimizing Mobile Applications.

While the Unite talk implied that the covered topics might only be relevant to mobile performance, much of the talk was applicable to a general audience. Effort has been taken in this write-up to discuss information related to performance in a broad and platform-independent manner.

Contents

Profiling
Memory
Coroutines
Asset auditing
Strings and text
The resources folder
General optimizations
Special optimizations

Profiling

Leave feedback

When discussing performance, it is vital to remember that all optimization attempts must start with a discovery process. Profiling an application to discover its hotspots is the necessary first step, followed by an analysis of profiling results against the project's technical and **Asset** architecture.

Note: Where this section discusses method names found in native-code profiling traces, the method names have been drawn from Unity 5.3. Method names may change in future versions of Unity.

Tools

For profiling, there are many different tools available to Unity developers. Unity has a suite of built-in tools, such as the CPU Profiler, Memory Profiler and the new 5.3 Memory Analyzer.

However, the best data generally comes from platform-specific tools. These include:

For **iOS**: **Instruments** and the **XCode Frame Debugger**

For Android: the **Snapdragon Profiler**

For platforms running Intel CPUs/GPUs: **VTune** and **Intel GPA**

For **PS4**: the **Razor** suite and **VR Trace**

For Xbox: the **Pix** tool

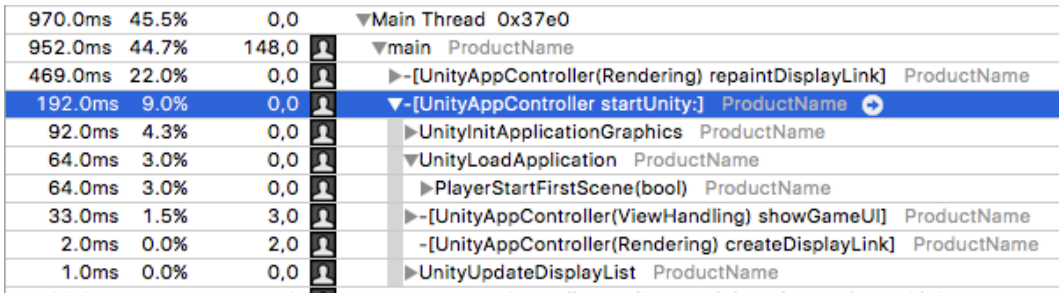
These tools generally have the most utility on platforms that can utilize IL2CPP to produce a C++ version of the project. These native-code versions provide transparent callstacks and high-resolution method timings that are unavailable when running under Mono.

Unity has already created a basic guide to using Instruments to profile iOS games; see Profiling with Instruments.

Dissecting startup traces

When looking at a trace of start-up times, there are two key methods to inspect. These two methods are the primary places where the configuration, Assets and code of a project can impact start-up time.

Note that start-up time manifests differently on different platforms. On most platforms it is visible to the user as a static splash screen.



970.0ms	45.5%	0,0	▼Main Thread 0x37e0
952.0ms	44.7%	148,0	▼main ProductName
469.0ms	22.0%	0,0	▶-[UnityAppController(Rendering) repaintDisplayLink] ProductName
192.0ms	9.0%	0,0	▼-[UnityAppController startUnity:] ProductName
92.0ms	4.3%	0,0	▶UnityInitApplicationGraphics ProductName
64.0ms	3.0%	0,0	▼UnityLoadApplication ProductName
64.0ms	3.0%	0,0	▶PlayerStartFirstScene(bool) ProductName
33.0ms	1.5%	3,0	▶-[UnityAppController(ViewHandling) showGameUI] ProductName
2.0ms	0.0%	2,0	-[UnityAppController(Rendering) createDisplayLink] ProductName
1.0ms	0.0%	0,0	▶UnityUpdateDisplayList ProductName

The above screenshot is from an Instruments trace of an example project running on an iOS device. Within the platform-specific `startUnity` method, note the `UnityInitApplicationGraphics` and `UnityLoadApplication` methods.

`UnityInitApplicationGraphics` performs a lot of internal work, such as setting up the graphics device and initializing many of Unity's internal systems. In addition, it initializes the Resources system. To do this, it must load an index of all of the files contained by the Resources system.

Every Asset file within every folder named "Resources" (1) (NOTE: This only applies to folders named "Resources" that are within the project's "Assets" folder, as well as all child folders within those "Resources" folders.) is included in the Resource system's data.

Therefore, the time required to initialize the Resources system increases at least linearly in correlation with the number of files within “Resources” folders.

`UnityLoadApplication` contains methods that load and initialize the first **Scene** in the project. This includes deserializing and instantiating all of the data necessary to display the first **Scene**, such as compiling **Shaders**, uploading Textures and instantiating **GameObjects**. Additionally, all MonoBehaviours in the first **Scene** have their `Awake` callbacks executed at this time.

These processes mean that if there is any long-running code in an `Awake` callback in the first Scene of a project, that code could be responsible for slowing down the project’s initial start-up time. Resolving this involves either eliminating the slow code, or executing it elsewhere in the application’s lifecycle.

Dissecting run time traces

For profiling traces captured after initial startup time, the primary place of interest is the method `PlayerLoop`. This is Unity’s main loop, and the code within it runs once per frame.

970.0ms	45.5%	0,0	▼Main Thread 0x37e0
952.0ms	44.7%	148,0	▼main ProductName
469.0ms	22.0%	0,0	▼-[UnityAppController(Rendering) repaintDisplayLink] ProductName
464.0ms	21.8%	1,0	▼UnityRepaint ProductName
457.0ms	21.4%	0,0	▼UnityPlayerLoopImpl(bool) ProductName
457.0ms	21.4%	1,0	▼PlayerLoop(bool, bool, IHookEvent*) ProductName
212.0ms	9.9%	0,0	▼PostLateUpdate_FinishFrameRendering ProductName
210.0ms	9.8%	0,0	▶PlayerRender(bool) ProductName
1.0ms	0.0%	1,0	IsBatchmode() ProductName
1.0ms	0.0%	0,0	▶PresentAfterDraw(GfxDevice::PresentMode) ProductName
102.0ms	4.7%	2,0	▶PhysicsManager::FixedUpdate() ProductName
30.0ms	1.4%	1,0	▶DirectorManager::ExecuteStage(DirectorStage) ProductName
18.0ms	0.8%	0,0	▶NavMeshManager::Update() ProductName
16.0ms	0.7%	0,0	▶PostLateUpdate_UpdateAudio ProductName
11.0ms	0.5%	0,0	▶PostLateUpdate_UpdateAllSkinnedMeshes ProductName
10.0ms	0.4%	0,0	▶Script_RunDelayedFixedFrameRate ProductName
8.0ms	0.3%	0,0	▼Script_RunBehaviourUpdate ProductName
8.0ms	0.3%	0,0	▶void BaseBehaviourManager::CommonUpdate<BehaviourManager>() ProductName
6.0ms	0.2%	0,0	▼Script_RunBehaviourFixedUpdate ProductName
6.0ms	0.2%	0,0	▶void BaseBehaviourManager::CommonUpdate<FixedBehaviourManager>() ProductName
6.0ms	0.2%	0,0	▶FixedUpdate_AudioFixedUpdate ProductName
5.0ms	0.2%	0,0	▶PostLateUpdate_PlayerUpdateCanvases ProductName

The above screenshot is from a profiling run of an example Unity 5.4 project, and illustrates several of the most interesting methods within `PlayerLoop`. Note that the names of methods within the `PlayerLoop` may vary between Unity versions.

`PlayerRender` is the method that runs Unity’s rendering system. This includes culling objects, calculating dynamic batches, and submitting drawing instructions to the GPU. Any Image Effects or rendering-based script callbacks (`OnWillRenderObject`, for example) also run here. In general, this should be the top consumer of CPU time while the project is interactive.

`BaseBehaviourManager` calls three templated versions of `CommonUpdate`. These invoke certain callbacks within MonoBehaviours attached to active GameObjects in the current Scene.

`CommonUpdate<UpdateManager>` calls `Update` callbacks

`CommonUpdate<LateUpdateManager>` calls `LateUpdate` callbacks

`CommonUpdate<FixedUpdateManager>` calls `FixedUpdate` if the physics system has ticked

In general, `BaseBehaviourManager::CommonUpdate<UpdateManager>` is the most interesting method family to inspect, because it is the entry point for most of the script code running within a Unity project.

There are several other methods of interest:

`UI::CanvasManager` invokes several different callbacks if a project uses Unity UI. This includes Unity UI's batch computation and layout updates; the two operations that most often cause `CanvasManager` to appear in the profiler.

`DelayedCallManager::Update` runs coroutines. These are covered in more detail in the "Coroutines" chapter of this document.

`PhysicsManager::FixedUpdate` runs the PhysX physics system. This primarily involves running PhysX's internal code, and is influenced by the number of physics objects in the current Scene, such as Rigidbodies and Colliders. However, physics-based callbacks also appear here – in particular, `OnTriggerStay` and `OnCollisionStay`.

If the project is using 2D physics, that appears as an analogous set of calls under `Physics2DManager::FixedUpdate`.

Dissecting a script method

When scripts are invoked on platforms cross-compiled with IL2CPP, look for trace lines containing a `ScriptingInvocation` object. This is the point where Unity's internal native code is transitioning into the script runtime in order to execute script code (2) (NOTE: Technically, after being run through IL2CPP, C#/JS script code also becomes native code. However, this cross-compiled code primarily executes methods via the IL2CPP runtime framework and does not closely resemble handwritten C++.).

67.0ms	2.1%	0,0	▼Script_RunBehaviourUpdate ProductName
67.0ms	2.1%	0,0	▼void BaseBehaviourManager::CommonUpdate<BehaviourManager>() ProductName
66.0ms	2.1%	1,0	▼UpdateBehaviour ProductName
64.0ms	2.0%	0,0	▼MonoBehaviour::CallUpdateMethod(int) ProductName
63.0ms	2.0%	0,0	▼CallMethodIfAvailable ProductName
63.0ms	2.0%	0,0	▼ScriptingInvocationNoArgs::Invoke() ProductName
63.0ms	2.0%	0,0	▼ScriptingInvocationNoArgs::Invoke(ScriptingException**) ProductName
61.0ms	1.9%	0,0	▼0x100d3badc ProductName
61.0ms	1.9%	0,0	▼RuntimeInvoker_Void_t2863195528(MethodInfo const*, void*, void**) ProductName
24.0ms	0.7%	0,0	▶StandaloneInputModule_Process_m3720469665 ProductName
17.0ms	0.5%	0,0	▶EventSystem_Update_m242895889 ProductName
13.0ms	0.4%	0,0	▶PlayerShooting_Update_m2128394500 ProductName
3.0ms	0.0%	0,0	▶ScoreManager_Update_m318092571 ProductName
3.0ms	0.0%	0,0	▶PlayerShooting_DisableEffects_m2460645109 ProductName
1.0ms	0.0%	0,0	▶Transform_Translate_m1056984957 ProductName

The above screenshot is from another trace from an example project running in Unity 5.4. All of the methods nested beneath the `RuntimeInvoker_Void` line are part of cross-compiled C# scripts that are being executed once per frame.

The trace lines are fairly easy to read: each one is the name of the original class followed by an underscore and the name of the original method. In this example trace, it's possible to see `EventSystem.Update`, `PlayerShooting.Update` and several other `Update` methods. These are the standard Unity `Update` callbacks found in most MonoBehaviours.

By expanding these methods, it is possible to discover exactly which methods within them are consuming CPU time. This includes other script methods within the project, Unity APIs and C# library code.

The above trace shows that the `StandaloneInputModule.Process` method is raycasting through the entire UI once per frame, in order to detect whether any touch events are hovering over or activating any UI elements. The primary cost is iterating over all the UI elements, and testing whether the mouse's position is within their bounding rectangle.

Asset loading

Asset loading can also be identified in CPU traces. The main method indicating an Asset load is `SerializedFile::ReadObject`. This method connects a binary data stream (from a file) to Unity's serialization system, which operates via a method named `Transfer`. The `Transfer` method can be found on all Asset types, such as Textures, MonoBehaviours and Particle Systems.

62.0ms	1.0%	0,0	▼LoadSceneOperation::Perform() sampleassets
61.0ms	1.0%	0,0	▼PersistentManager::LoadFileCompletelyThreaded(std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<char> >
61.0ms	1.0%	0,0	▼SerializedFile::ReadObject(long long, ObjectCreationMode, bool, TypeTree const**, bool*, Object&) sampleassets
23.0ms	0.3%	0,0	▶void GameObject::Transfer<StreamedBinaryRead<false> >(StreamedBinaryRead<false>&) sampleassets
16.0ms	0.2%	0,0	▶void MonoBehaviour::TransferEngineAndInstance<StreamedBinaryRead<false> >(StreamedBinaryRead<false>&) sampleassets
14.0ms	0.2%	0,0	▶void Mesh::Transfer<StreamedBinaryRead<false> >(StreamedBinaryRead<false>&) sampleassets
3.0ms	0.0%	0,0	▶void Shader::Transfer<StreamedBinaryRead<false> >(StreamedBinaryRead<false>&) sampleassets
3.0ms	0.0%	0,0	▶void TextRenderingPrivate::Font::Transfer<StreamedBinaryRead<false> >(StreamedBinaryRead<false>&) sampleassets
1.0ms	0.0%	1,0	void LightmapSettings::Transfer<StreamedBinaryRead<false> >(StreamedBinaryRead<false>&) sampleassets
1.0ms	0.0%	0,0	▶void Sprite::Transfer<StreamedBinaryRead<false> >(StreamedBinaryRead<false>&) sampleassets

In the above screenshot, a Scene is being loaded. This requires Unity to read and deserialize all the Assets within the Scene, as denoted by the calls to various `Transfer` methods beneath `SerializedFile::ReadObject`.

In general, if a performance stutter is seen during runtime and a performance trace shows significant time being used by `SerializedFile::ReadObject`, the framerate is being reduced due to Asset loads. Note that, in most cases, `SerializedFile::ReadObject` can be found on the main thread only when synchronous Asset loads are requested via the `SceneManager`, `Resources` or AssetBundle APIs.

This sort of performance stutter can be remediated in the usual ways: you can make Asset loading asynchronous (which moves the heavy `ReadObject` call to a worker thread), or preload certain heavy Assets.

Note that `Transfer` calls also appear when cloning objects (denoted by the `CloneObject` method in a trace). If a call to `Transfer` appears beneath a `CloneObject` call, then the Asset is not being loaded from storage. Instead, the old Object's data is being transferred to the new Object. To do this, Unity serializes the old Object and deserializes the resulting data as the new Object.

Footnotes

(1) This only applies to folders named *Resources* that are within the project's *Assets* folder, as well as all child folders within those *Resources* folders.
(2) Technically, after being run through IL2CPP, C#/JS script code also becomes native code. However, this cross-compiled code primarily executes methods via the IL2CPP runtime framework and does not closely resemble handwritten C++.

2018–04–25 Page amended with limited editorial review

Memory

Leave feedback

Memory consumption is a critical performance indicator, and is especially important on platforms with limited memory resources, such as low-end mobile devices.

Profiling memory consumption

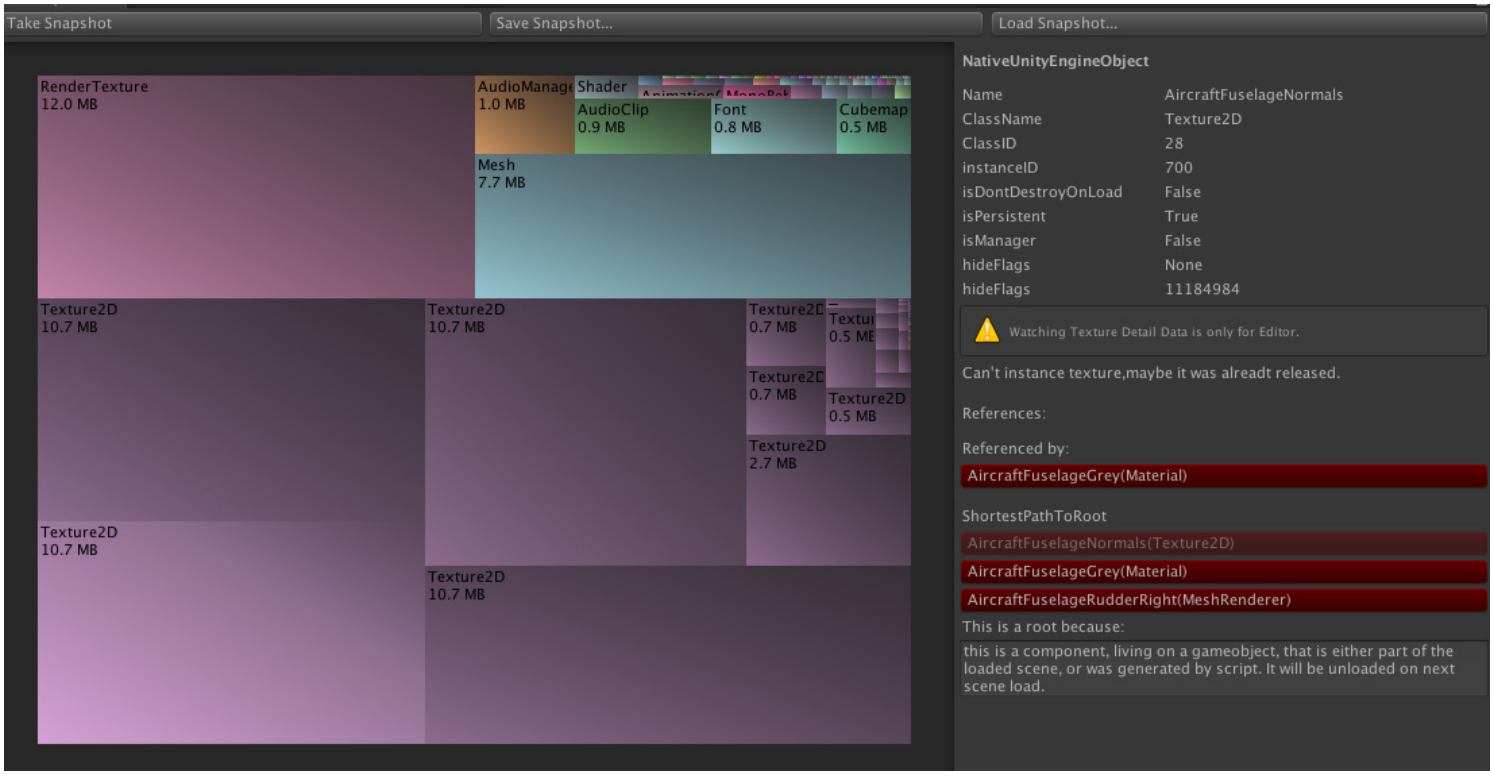
Diagnosing memory issues in Unity is best performed with an open-source memory visualization tool available from Unity's Bitbucket. Integration of this tool is as simple as downloading the linked repository and dropping the included *Editor* folder into a project.

The tool can be used with any version of Unity from 5.3 onwards. It can capture a wealth of information on native- and managed-code memory consumption when attached to an application built with **IL2CPP**.

To use the tool, simply build a project with an IL2CPP scripting backend and deploy it to the appropriate device. Attach Unity's regular in-editor CPU profiler, then open the Memory Profiler window (menu: **Window** > **MemoryProfilerWindow**) and selet **Take Snapshot**.

The application on device pauses for a short time as data is gathered and transmitted to the Unity Editor. Then, the Unity Editor pauses to parse the received data; this can require a significant amount of time. For very memory-intensive applications, a trace might require 10–30 minutes to be parsed.

During this parse-and-load operation, patience is recommended.



This screenshot is from a **Standard Assets Scene** running on an **iOS** device, and shows that over three-quarters of memory usage is attributable to four very large Textures, all related to the aircraft's fuselage.

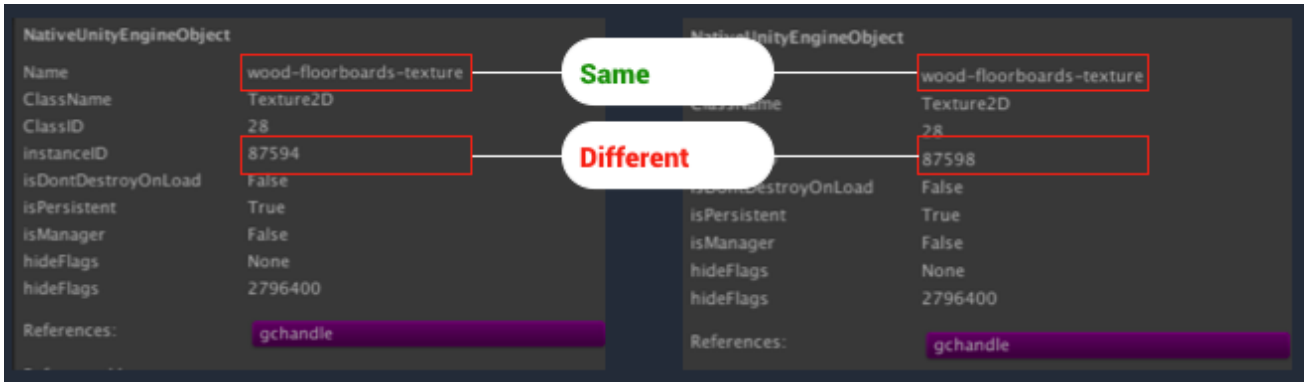
This visualization can be zoomed. Click each box in the application to see more information about it.

Identifying duplicated Textures

One common memory problem is the duplication of **Assets** in memory. As Textures are often the most memory-intensive **Assets** in a project, Texture duplication is one of the most common memory problems found in Unity projects.

Duplicated Assets can be identified by locating two objects of the same type and same size, which appear to have been loaded from the same Asset. In the detail pane of the new Memory **Profiler**, examine the **Name** and **InstanceID** fields of the seemingly-identical objects.

The **Name** field is based on the name of the Asset file from which the object was loaded; generally, it is the file name without the file's path or extension. The **InstanceID** field indicates the internal identification number assigned by the Unity runtime; this number is unique during a single run of a Unity game(1).



This diagram demonstrates a simple example of this problem. On the left and right sides of the diagram are screenshots taken from the Detail pane of the 5.4 Memory Profiler. The Assets shown in these screenshots are two Textures loaded separately in memory. The Textures have identical names and sizes, suggesting that they may be duplicates. By examining the project's "Assets" folder, it can be determined that there is only one Asset file named *wood-floorboards-texture*, which very strongly points towards asset duplication.

Each individual UnityEngine.Object in memory has a unique instance ID, assigned to it when the object is created. As these two Textures have different instance IDs, it is certain that they represent two different sets of Texture data that are loaded into memory.

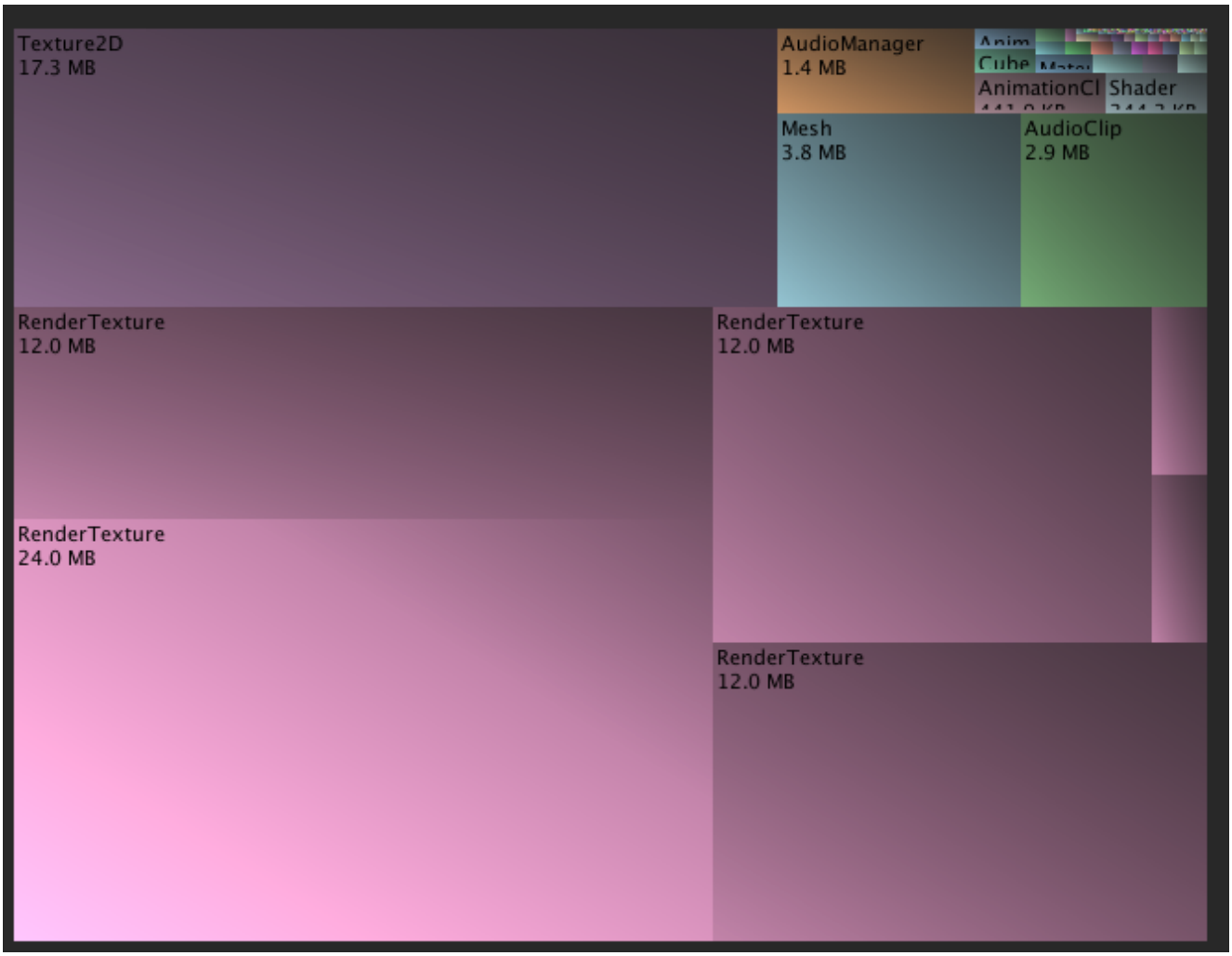
As the file names and Asset sizes are identical, while the instance IDs are different, it is certain that these two objects represent a Texture that has been duplicated in memory (NOTE: If there are Textures with identical filenames in the project, then this judgment is not absolute, but still strongly suggestive when correlated with the identical file sizes.).

AssetBundles and Asset duplication

The most common cause of Texture and Asset duplication in memory is the improper unloading of AssetBundles. See Unity's Best Practice Guide on AssetBundles for a description of this problem. The key section is Managing Loaded Assets.

Examining image buffer, Image Effect & RenderTexture memory usage

It is also possible to visualize the memory required to provide render buffers to Image Effects and RenderTexture objects in the memory visualizer.



The above screenshot demonstrates a simple Scene with a handful of Unity's Cinematic Image Effects applied to it. The Image Effects allocate temporary render buffers in order to perform their computations; in particular, the Bloom effect allocates several buffers of decreasing size. Due to the high resolution of Retina iOS devices, these temporary buffers consume significantly more memory than the rest of the project combined.

Consider that an iPad Air 2 renders at 2048x1536 – this is beyond the 1080p resolution often targeted on modern consoles and PCs, but is running on a tablet device. A full-screen temporary render buffer consumes a full 24 or 36 megabytes of memory, depending on the format of the buffer. This can be reduced by 75% by halving the **pixel** dimensions of the render buffer. This often does not significantly degrade the visual quality of the result.

One way to optimize Image Effects' use of temporary render buffers and other GPU resources is to create a single "uber" Image Effect that performs all of the disparate calculations at once. When using Unity 5.5 or newer, it is possible to use the new UberFX (available from github package; this package provides a configurable "uber" Image Effect that can perform all of the operations provided by the Cinematic Image Effects with less overhead than individual Image Effects.

Coroutines

Leave feedback

Coroutines execute differently from other script code. Most script code simply appears within a performance trace in a single location, beneath a specific Unity callback invocation. However, the CPU code of coroutines always appears in two places in a trace.

All of the initial code in a coroutine, from the start of the coroutine method until it yields for the first time, appears in the trace wherever the coroutine is started. Usually, it appears wherever the `StartCoroutine` method is called. Coroutines generated from Unity callbacks (such as `Start` callbacks that return an `IEnumerator`) first appear within their respective Unity callback.

All of the rest of a coroutine's code – from the first time it resumes until it finished executing – appears within the `DelayedCallManager` line that appears inside Unity's main loop.

To understand why this occurs, consider how a coroutine is actually executed.

Coroutines are backed by an instance of a class that is autogenerated by the C# compiler. This object is needed to track the state of the coroutine across multiple invocations of what is, to the programmer, a single method. Because local-scope variables within the coroutine must persist across `yield` calls, those local-scope variables are hoisted into the generated class and therefore remain allocated on the heap for the duration of the coroutine. This object also tracks the internal state of the coroutine: it remembers at which point in the code the coroutine must be resume after yielding.

Because of this, the memory pressure caused by starting a coroutine is equal to a fixed overhead cost plus the size of its local-scope variables.

The code which starts a coroutine constructs and invokes this object, and then Unity's `DelayedCallManager` invokes it again whenever the coroutine's yield condition has been satisfied. As coroutines usually start outside of other coroutines, this splits the cost of their execution into the two locations described above.

17.0ms	0.2%	0,0	▼Script_RunDelayedDynamicFrameRate servicerunnerTest
17.0ms	0.2%	0,0	▼DelayedCallManager::Update(int) servicerunnerTest
15.0ms	0.1%	1,0	▼Coroutine::Run() servicerunnerTest
13.0ms	0.1%	1,0	▼Coroutine::InvokeMoveNext(ScriptingException**) servicerunnerTest
10.0ms	0.1%	0,0	▼Invoke servicerunnerTest
10.0ms	0.1%	0,0	▼ScriptingInvocation::Invoke(ScriptingException**, bool) servicerunnerTest
10.0ms	0.1%	0,0	▼0x100db2430 servicerunnerTest
10.0ms	0.1%	0,0	▼RuntimeInvoker_Void_t2863195528_Il2CppObject_IntPtr_t(MethodInfo const*, void*, void**)
10.0ms	0.1%	1,0	▼SetupCoroutine_InvokeMoveNext_m3556339879 servicerunnerTest
3.0ms	0.0%	0,0	▶U3CPopulateCharactersU3Ec__Iterator7_MoveNext_m3361908547 servicerunnerTest
2.0ms	0.0%	0,0	▶U3CAsyncLoadU3Ec__Iterator0_MoveNext_m1879388671 servicerunnerTest
2.0ms	0.0%	0,0	▶U3CLoadDatabaseU3Ec__Iterator4_MoveNext_m3893571312 servicerunnerTest
1.0ms	0.0%	0,0	▶U3CLoadDatabaseU3Ec__Iterator8_MoveNext_m3614029908 servicerunnerTest

This can be observed in the above screenshot, where the `DelayedCallManager` is resuming several different coroutines: `PopulateCharacters`, `AsyncLoad` and `LoadDatabase` are the notable ones.

When possible, it is better to condense a series of operations down to the fewest number of individual coroutines possible. While nested coroutines are excellent for code clarity and maintenance, they impose a higher memory overhead due to the coroutine tracking objects.

If a coroutine runs nearly every frame and does not yield on long-running operations, it is generally more readable to replace it with an `Update` or `LateUpdate` callback. This is particularly true of long-running or infinitely-looping coroutines.

Coroutines are not stopped when an object is disabled, but only when it is definitely destroyed. This allows coroutines to still run and, if needed, enable the object again, for example. Calling Destroy(this) triggers `OnDisable` immediately and the coroutines are processed. Finally, `OnDestroy` is invoked at the end of the frame.

It is important to remember that coroutines are *not* threads. Synchronous operations running within a coroutine still execute on the main thread. If the goal is to reduce CPU time spent on the main thread, it is just as important to avoid blocking operations in coroutines as in any other script code.

Coroutines are best employed when dealing with long asynchronous operations, such as waiting for HTTP transfers, **Asset** loads or file I/O to complete.

Asset auditing

Leave feedback

Many of the problems found in real projects occur because of honest mistakes – temporary "test" changes and misclicks by a tired developer can silently add poorly-performing **Assets**, or change the import settings of existing **Assets**.

For any project of significant scale, it is best to have a first line of defense against human errors. It is relatively simple to write a small piece of code that ensures that no one can add a 4K uncompressed Texture to a project.

And yet, this is a surprisingly common problem. A 4K uncompressed Texture occupies 60 megabytes of memory. On a low-end mobile device, such as an iPhone 4S, it is dangerous to consume more than about 180–200 megabytes of memory. If added by mistake, this Texture inadvertently occupies between a third and a quarter of the application's memory budget, and causes difficult-to-diagnose out-of-memory errors.

While it is now possible to track these issues down with the 5.3 Memory **Profiler**, it is arguably better to make sure such mistakes are impossible in the first place.

Using AssetPostprocessor

The `AssetPostprocessor` class in the Unity Editor can be used to enforce certain minimum standards on a Unity project. This class receives callbacks when Assets are imported. To use it, inherit from `AssetPostprocessor` and implement one or more `OnPreprocess` methods. Significant ones include:

`OnPreprocessTexture`

`OnPreprocessModel`

`OnPreprocessAnimation`

`OnPreprocessAudio`

See the Scripting Reference on AssetPostprocessor for more possible `OnPreprocess` methods.

```
public class ReadOnlyModelPostprocessor : AssetPostprocessor {

   public void OnPreprocessModel() {

        ModelImporter modelImporter =

 (ModelImporter)assetImporter;

        if(modelImporter.isReadable) {

            modelImporter.isReadable = false;

            modelImporter.SaveAndReimport();
```

```
            }

        }

    }
```

This is a simple example of an `AssetPostprocessor` enforcing rules on a project:

This class is called any time a model is imported into the project, or whenever a model has its import settings changed. The code simply checks to see if the `Read/Write enabled` flag (represented by the `isReadable` property) is set to `true`. If it is, it forces the flag to be `false` and then saves and reimports the Asset.

Note that calling `SaveAndReimport` causes this snippet of code to be called again! However, because it is now assured that `isReadable` is false, this code does not produce an infinite reimport loop.

The reason for this change is discussed in the “Models” section, below.

Common Asset rules

Textures

Disable the read/write enabled flag

The `Read/Write enabled` flag causes a Texture to be kept twice in memory: once on the GPU and once in CPU-addressable memory(1) (NOTE: This is because, on most platforms, readback from GPU memory is extremely slow. Reading a Texture from GPU memory into a temporary buffer for use by CPU code (e.g. Texture.GetPixel) would be very nonperformant.). In Unity, this setting is disabled by default, but it can be accidentally switched on.

`Read/Write Enabled` is only necessary when manipulating Texture data outside of Shaders (such as with the `Texture.GetPixel` and `Texture.SetPixel` APIs) and should be avoided whenever possible.

Disable Mipmaps if possible

For objects that have a relatively invariant Z-depth relative to the Camera, it is possible to disable mipmaps to save about a third of the memory required to load the Texture. If an object changes Z-depth, disabling mipmaps can lead to worse Texture sampling performance on the GPU.

In general, this is useful for UI Textures and other Textures that are displayed at a constant size on screen.

Compress all Textures

Using a texture compression format suitable for the project’s target platform is crucial for saving memory.

If the selected texture compression format is unsuited to the target platform, Unity decompresses the Texture when the Texture is loaded, consuming both CPU time and an excessive amount of memory. This is most

commonly a problem on Android devices, which often support vastly different texture compression formats depending on the chipset.

Enforce sensible Texture size limits

While this is simple, it is also very easy to forget to resize a Texture or to inadvertently alter the Texture size import setting. Determine a sensible maximum size for different types of Textures and enforce these via code.

For many mobile applications, 2048x2048 or 1024x1024 is sufficient for Texture atlases, and 512x512 is sufficient for Textures applied to 3D models.

Models

Disable the Read/Write enabled flag

The`Read/Write enabled` flag for models operates identically to the one described for Textures. However, it is enabled by default for models.

Unity requires this flag to be enabled if a project is modifying a Mesh at runtime via script, or if the Mesh is used as the basis for a MeshCollider component. If the model is not used in a MeshCollider and is not manipulated by scripts, disable this flag to save half of the model's memory.

Disable rigs on non-character models

By default, Unity imports a generic rig for non-character models. This causes an `Animator` component be to added if the model is instantiated at runtime. If the model is not animated via the Animation system, this adds unnecessary overhead to the animation system, because all active Animators must be ticked once per frame.

Disable the rig on non-animated models to avoid this automatic addition of an Animator component and possible inadvertent addition of unwanted Animators to a Scene.

Enable the Optimize Game Objects option on animated models

The **Optimize Game Objects** option has a significant performance impact on animated models. When the option is disabled, Unity creates a large transform hierarchy mirroring the model's bone structure whenever the model is instantiated. This transform hierarchy is expensive to update, especially if other components (such as Particle Systems or Colliders) are attached to it. It also limits Unity's ability to multithread Mesh skinning and bone animation calculations.

If specific locations on a model's bone structure need to be exposed (such as exposing a model's hands in order to dynamically attach weapon models) then these locations can be specifically whitelisted in the `Extra Transforms` list.

Some additional details can be found in the Unity Manual page on the Model Importer.

Use Mesh compression when possible

Enabling Mesh compression reduces the number of bits used to represent the floating-point numbers for different channels of a model's data. This can lead to a minor loss of precision, and the effects of this imprecision should be checked by artists before use in a final project.

The specific numbers of bits that a given compression level uses are detailed in the ModelImporterMeshCompression Scripting Reference.

Note that it is possible to use different levels of compression for different channels, so a project may choose to compress only tangents and normals while leaving UVs and vertex positions uncompressed.

Note: Mesh Renderer Settings

When adding Mesh Renderers to Prefabs or GameObjects, take note of the settings on the component. By default, Unity enables Shadow casting and receiving, Light Probe sampling, Reflection Probe sampling, and Motion Vector calculation.

If a project does not require one or more of these features, ensure that they're turned off by an automated script. Any runtime code that adds MeshRenderers needs to toggle these settings as well.

For 2D games, accidentally adding a MeshRenderer to the Scene with the shadow options turned on adds a full shadow pass to the rendering loop. This is generally a waste of performance.

Audio

Platform-appropriate compression settings

Enable a compression format for audio that matches the available hardware. All iOS devices include hardware MP3 decompressors, and many Android devices support Vorbis natively.

Further, import uncompressed audio files into Unity. Unity always recompresses audio when building a project. There is no need to import compressed audio and then recompress it; this only serves to degrade the quality of the final audio clips.

Force audio clips to mono

Few mobile devices actually have stereo speakers. On a mobile project, forcing the imported audio clips to be mono-channel halves their memory consumption. This setting is also applicable to any audio without stereo effects, such as most UI sound effects.

Reduce audio bitrate

While this requires consultation with an audio designer, attempt to minimize the bitrate of audio files in order to further save on memory consumption and built project size.

Understanding the managed heap

Leave feedback

Another common problem faced by many Unity developers is the unexpected expansion of the managed heap. In Unity, the managed heap expands much more readily than it shrinks. Furthermore, Unity's garbage collection strategy tends to fragment memory, which can prevent a large heap from shrinking.

How the managed heap operates and why it expands

The "managed heap" is a section of memory that is automatically managed by the memory manager of a Project's scripting runtime (Mono or IL2CPP). All objects created in managed code must be allocated on the managed heap(2) (**Note:** Strictly speaking, all non-null reference-typed objects and all boxed value-typed objects must be allocated on the managed heap).

In the above diagram, the white box represents a quantity of memory apportioned to the managed heap, and the colored boxes within it represent data values stored within the managed heap's memory space. When additional values are needed, more space is allocated from within the managed heap.

The garbage collector runs periodically(3) (**Note:** The exact timing is platform-dependent). This sweeps through all objects on the heap, marking for deletion any objects that are no longer referenced. Unreferenced objects are then deleted, freeing up memory.

Crucially, Unity's garbage collection – which uses the Boehm GC algorithm – is non-generational and non-compacting. "Non-generational" means that the GC must sweep through the entire heap when performing a collection pass, and its performance therefore degrades as the heap expands. "Non-compacting" means that objects in memory are not relocated in order to close gaps between objects.

The above diagram shows an example of memory fragmentation. When an object is released, its memory is freed. However, the freed space does **not** become part of a single large pool of "free memory". The objects on either side of the freed object may still be in use. Because of this, the freed space is a "gap" between other segments of memory (this gap is indicated by the red circle in the diagram). The newly-freed space can therefore only be used to store data of identical or lesser size than the freed object.

When allocating an object, remember that the object must always occupy a contiguous block of space in memory.

This leads to the core problem of memory fragmentation: while the overall amount of space available in the heap may be substantial, it is possible that some or all of that space is in small "gaps" between allocated objects. In this

case, even though there may be enough total space to accommodate a certain allocation, the managed heap cannot find a large enough block of contiguous memory in which to fit the allocation.

However, if a large object is allocated and there is insufficient contiguous free space to accommodate the object, as illustrated above, the Unity memory manager performs two operations.

First, if it has not already done so, the garbage collector runs. This attempts to free up enough space to fulfill the allocation request.

If, after the GC runs, there is still not enough contiguous space to fit the requested amount of memory, the heap must expand. The specific amount that the heap expands is platform-dependent; however, most Unity platforms double the size of the managed heap.

Key problems with the heap

The core issues with managed heap expansion are twofold:

Unity does not often release the memory pages allocated to the managed heap when it expands; it optimistically retains the expanded heap, even if a large portion of it is empty. This is to prevent the need to re-expand the heap should further large allocations occur.

On most platforms, Unity eventually releases the pages used by empty portions of the managed heap back to the operating system. The interval at which this occurs is not guaranteed and should not be relied upon.

The address space used by the managed heap is never returned to the operating system.

For 32-bit programs, this can lead to address space exhaustion if the managed heap expands and contracts many times. If a program's available memory address space is exhausted, the operating system will terminate the program.

For 64-bit programs, the address space is sufficiently large that this is extremely unlikely to occur for programs whose running time does not exceed the average human lifespan.

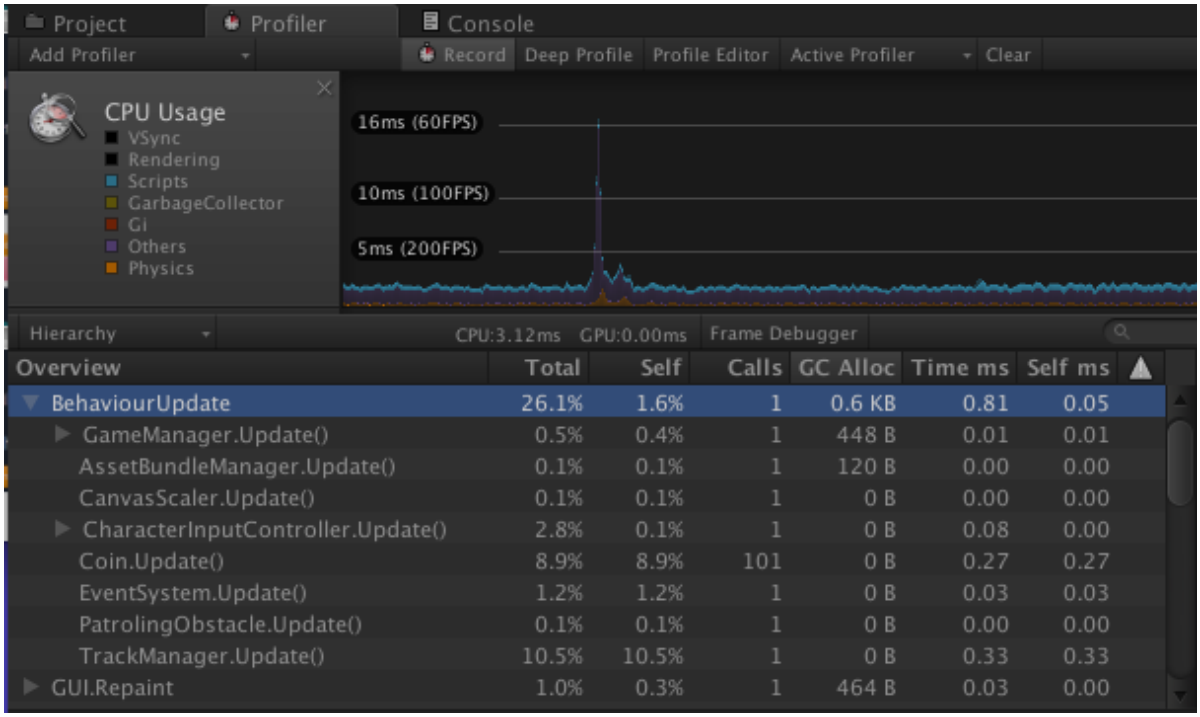
Temporary allocations

Many Unity projects are found to operate with several tens or hundreds of kilobytes of temporary data being allocated to the managed heap each frame. This is often extremely detrimental to a project's performance. Consider the following math:

If a program allocates one kilobyte (1kb) of temporary memory each frame, and is running at 60 **frames per second**, then it must allocate 60 kilobytes of temporary memory per second. Over the course of a minute, this adds up to 3.6 megabytes of garbage in memory. Invoking the garbage collector once per second is

likely to be detrimental to performance, but allocating 3.6 megabytes per minute is problematic when attempting to run on low-memory devices.

Further, consider loading operations. If a large number of temporary objects are generated during a heavy Asset-loading operation, and those objects are referenced until the operation completes, then the garbage collector is unable to release those temporary objects and the managed heap needs to expand – even though many of the objects it contains will be released a short time later.



Keeping track of managed memory allocations is relatively simple. In Unity's CPU **Profiler**, the Overview has a "GC Alloc" column. This column displays the number of bytes allocated on the managed heap in a specific frame (4) (**Note:** Note that this is not identical to the number of bytes temporarily allocated during a given frame. The profile displays the number of bytes allocated in a specific frame, even if some/all of the allocated memory is reused in subsequent frames). With the "Deep Profiling" option enabled, it's possible to track down the method in which these allocations occur.

The Unity Profiler does not track these allocations when they occur off the main thread. Therefore, the "GC Alloc" column cannot be used to measure managed allocations that occur in user-created threads. Switch the execution of code from separate threads to the main thread for debugging purposes or use the BeginThreadProfiling API to display the samples in the **Timeline** Profiler.

Always profile managed allocations with a **development build** on the target device.

Note that some script methods cause allocations when running in the Editor, but do not produce allocations after the project has been built. `GetComponent` is the most common example; this method always allocates when executed in the Editor, but not in a built project.

In general, it is strongly recommended that all developers minimize managed heap allocations whenever the project is in an interactive state. Allocations during non-interactive operations, such as **Scene** loading, are less problematic.

The Jetbrains Resharper Plugin for Visual Studio can help locate allocations in code.

Use Unity's Deep Profile mode to locate the specific causes of managed allocations. In Deep Profile mode, all method calls are recorded individually, providing a clearer view of where managed allocations occur within the method call tree. Note that Deep Profile mode works not only in the Editor but also on Android and Desktop using the command line argument `-deepprofiling`. The Deep Profiler button stays grayed out during profiling.

Basic memory conservation

There are a handful of relatively simple techniques that can be employed to reduce managed heap allocations.

Collection and array reuse

When using C#'s Collection classes or Arrays, consider reusing or pooling the allocated Collection or Array whenever possible. The Collection classes expose a *Clear* method which eliminates the Collection's values but does not release the memory allocated to the Collection.

```
void Update() {

    List<float> nearestNeighbors = new List<float>();

    findDistancesToNearestNeighbors(nearestNeighbors);

    nearestNeighbors.Sort();

    // … use the sorted list somehow …

}
```

This is particularly useful when allocating temporary "helper" Collections for complex computations. A very simple example might be the following code:

In this example, the `nearestNeighbors` List is allocated once per frame in order to collect a set of data points. It's very simple to hoist this List out of the method and into the containing class, which avoids allocating a new List each frame:

```
List<float> m_NearestNeighbors = new List<float>();

void Update() {

    m_NearestNeighbors.Clear();

    findDistancesToNearestNeighbors(NearestNeighbors);
```

```
    m_NearestNeighbors.Sort();

    // … use the sorted list somehow …

}
```

In this version, the List's memory is retained and reused across multiple frames. New memory is only allocated when the List needs to expand.

Closures and anonymous methods

There are two points to consider when using closures and anonymous methods.

First, all method references in C# are reference types, and are therefore allocated on the heap. Temporary allocations can be easily created by passing a method reference as an argument. This allocation occurs regardless of whether the method being passed is an anonymous method or a predefined one.

Second, converting an anonymous method to a closure significantly increases the amount of memory required to pass the closure to method receiving it.

Consider the following code:

```
List<float> listOfNumbers = createListOfRandomNumbers();

listOfNumbers.Sort( (x, y) =>

(int)x.CompareTo((int)(y/2))

);
```

This snippet uses a simple anonymous method to control the sorting order of the list of numbers created on the first line. However, if a programmer wished to make this snippet reusable, it is tempting to substitute the constant `2` for a variable in local scope, like so:

```
List<float> listOfNumbers = createListOfRandomNumbers();

int desiredDivisor = getDesiredDivisor();
```

```
listOfNumbers.Sort( (x, y) =>

(int)x.CompareTo((int)(y/desiredDivisor))

);
```

The anonymous method now requires the method to be able to access the state of a variable outside of the method's scope, and so has become a closure. The `desiredDivisor` variable must be passed into the closure somehow so that it can be used by the actual code of the closure.

To do this, C# generates an anonymous class that can retain the externally-scoped variables needed by the closure. A copy of this class is instantiated when the closure is passed to the `Sort` method, and the copy is initialized with the value of the `desiredDivisor` integer.

Because executing the closure requires instantiation of a copy of its generated class, and all classes are reference types in C#, then executing the closure requires allocation of an object on the managed heap.

In general, it is best to avoid closures in C# whenever possible. Anonymous methods and method references should be minimized in performance-sensitive code, and especially in code that executes on a per-frame basis.

Anonymous methods under IL2CPP

Currently, inspection of code generated by **IL2CPP** reveals that the simple declaration and assignment of a variable of type `System.Function` allocates a new object. This is true whether the variable is explicit (declared in a method/class) or implicit (declared as an argument to another method).

As such, any use of anonymous methods under the IL2CPP **scripting backend** allocates managed memory. This is not the case under the Mono **scripting backend**.

Furthermore, IL2CPP displays dramatically different levels of managed memory allocation depending on the way in which a method argument is declared. Closures, as expected, allocate the most memory per call.

Unintuitively, predefined methods allocate **nearly as much memory as closures** when passed as arguments under the IL2CPP scripting backend. Anonymous methods generate the least amount of transient garbage on the heap, by one or more orders of magnitude.

Therefore, if a project is intended to ship on the IL2CPP scripting backend, there are three key recommendations:

Prefer coding styles that do not require passing methods as arguments.

When unavoidable, prefer anonymous methods over predefined methods.

Avoid closures, regardless of scripting backend.

Boxing

Boxing is one of the most common sources of unintended temporary memory allocations found in Unity projects. It occurs whenever a value-typed value is utilized as a reference type; this most often occurs when passing primitive value-typed variables (such as `int` and `float`) to object-typed methods.

In this extremely simple example, the integer in *x* is boxed in order to be passed to the `object.Equals` method, because the `Equals` method on `object` requires that an `object` be passed to it.

```
int x = 1;

object y = new object();

y.Equals(x);
```

C# IDEs and compilers generally do not issue warnings about boxing, even though it leads to unintended memory allocations. This is because the C# language was developed with the assumption that small temporary allocations would be efficiently handled by generational garbage collectors and allocation-size-sensitive memory pools.

While Unity's allocator does use different memory pools for small and large allocations, Unity's garbage collector is `not` generational and therefore cannot efficiently sweep out the small, frequent temporary allocations generated by boxing.

Boxing should be avoided wherever possible when writing C# code for Unity runtimes.

Identifying boxing

Boxing shows up in CPU traces as calls to one of a few methods, depending on the scripting backend in use. These generally take one of the following forms, where `<some class>` is the name of some other class or struct, and … is some number of arguments:

```
<some class>::Box(…)

Box(…)

<some class>_Box(…)
```

It can also be located by searching the output of a decompiler or IL viewer, such as the IL viewer tool built into ReSharper or the dotPeek decompiler. The IL instruction is "box".

Dictionaries and enums

One common cause of boxing is the use of `enum` types as keys for Dictionaries. Declaring an `enum` creates a new value type that is treated like an integer behind the scenes, but enforces type-safety rules at compile time.

By default, a call to `Dictionary.add(key, value)` results in a call to `Object.getHashCode(Object)`. This method is used to obtain the appropriate hash code for the Dictionary's key, and is used in all methods that accept a key: `Dictionary.tryGetValue, Dictionary.remove,` **etc**.

The `Object.getHashCode` method is reference-typed, but `enum` values are always value types. Therefore, for enum-keyed Dictionaries, every method call results in the key being boxed at least once.

The following code snippet illustrates a simple example that demonstrates this boxing problem:

```
enum MyEnum { a, b, c };

var myDictionary =

new Dictionary<MyEnum, object>();

myDictionary.Add(MyEnum.a, new object());
```

To solve this problem, it is necessary to write a custom class that implements the `IEqualityComparer` interface and assign an instance of that class as the Dictionary's comparer (**Note:** This object is usually stateless, and therefore can be reused with different Dictionary instances to save memory).

The following is a simple example of an IEqualityComparer for the above code snippet.

```
public class MyEnumComparer : IEqualityComparer<MyEnum> {

    public bool Equals(MyEnum x, MyEnum y) {

        return x == y;

    }

    public int GetHashCode(MyEnum x) {

        return (int)x;

    }

}
```

An instance of the above class could be passed to the Dictionary's constructor.

Foreach loops

In Unity's version of the Mono C# compiler, use of the `foreach` loop forces Unity to box a value each time the loop terminates (**Note:** The value is boxed once each time the loop as a whole finishes executing. It does not box once per iteration of the loop, so memory usage remains the same regardless of whether the loop runs two times or 200 times). This is because the IL generated by Unity's C# compiler constructs a generic value-type Enumerator in order to iterate over the value collection.

This Enumerator implements the `IDisposable` interface, which must be called when the loop terminates. However, calling interface methods on value-typed objects (such as structs and Enumerators) requires boxing them.

Examine the following very simple example code:

```
int accum = 0;

foreach(int x in myList) {

    accum += x;

}
```

The above, when run through Unity's C# compiler, produces the following Intermediate Language:

```
   .method private hidebysig instance void

    ILForeach() cil managed

  {

    .maxstack 8

    .locals init (

      [0] int32 num,

      [1] int32 current,

      [2] valuetype [mscorlib]System.Collections.Generic.List`1/Enumerator<int32>
```

```
)
// [67 5 - 67 16]
IL_0000: ldc.i4.0
IL_0001: stloc.0      // num
// [68 5 - 68 74]
IL_0002: ldarg.0      // this
IL_0003: ldfld        class [mscorlib]System.Collections.Generic.List`1<int32>
IL_0008: callvirt     instance valuetype [mscorlib]System.Collections.Generic
IL_000d: stloc.2      // V_2
.try
{
  IL_000e: br           IL_001f
// [72 9 - 72 41]
  IL_0013: ldloca.s     V_2
  IL_0015: call         instance !0/*int32*/ valuetype [mscorlib]System.Collec
  IL_001a: stloc.1      // current
// [73 9 - 73 23]
  IL_001b: ldloc.0      // num
  IL_001c: ldloc.1      // current
  IL_001d: add
  IL_001e: stloc.0      // num
// [70 7 - 70 36]
  IL_001f: ldloca.s     V_2
```

```
      IL_0021: call         instance bool valuetype [mscorlib]System.Collections.G

      IL_0026: brtrue       IL_0013

      IL_002b: leave        IL_003c

    } // end of .try

    finally

    {

      IL_0030: ldloc.2      // V_2

      IL_0031: box          valuetype [mscorlib]System.Collections.Generic.List`1/

      IL_0036: callvirt     instance void [mscorlib]System.IDisposable::Dispose()

      IL_003b: endfinally

    } // end of finally

    IL_003c: ret

  } // end of method test::ILForeach

} // end of class test
```

The most relevant code is the `__finally { … }__` block near the bottom. The `callvirt` instruction discovers the location of the `IDisposable.Dispose` method in memory before invoking the method, and requires that the Enumerator be boxed.

In general, `foreach` loops should be avoided in Unity. Not only do they box, but the method-call cost of iterating over collections via Enumerators is generally much slower than manual iteration via a `for` or `while` loop.

Note that the C# compiler upgrade in Unity 5.5 significantly improves Unity's ability to generate IL. In particular, the boxing operations has been eliminated from `foreach` loops. This eliminates the memory overhead associated with `foreach` loops. However, the CPU performance difference compared to equivalent Array-based code remains, due to method-call overhead.

Array-valued Unity APIs

A more pernicious and less-visible cause of spurious array allocation is the repeated accessing of Unity APIs that return arrays. All Unity APIs that return arrays create a new copy of the array each time they are accessed. It is

extremely non-optimal to access an array-valued Unity API more often than necessary.

As an example, the following code spuriously creates four copies of the `vertices` array per loop iteration. The allocations are occur each time the `.vertices` property is accessed.

```
for(int i = 0; i < mesh.vertices.Length; i++)
{
    float x, y, z;
    x = mesh.vertices[i].x;
    y = mesh.vertices[i].y;
    z = mesh.vertices[i].z;
    // ...
    DoSomething(x, y, z);
}
```

This can be trivially refactored into a single array allocation, regardless of the number of loop iterations, by capturing the `vertices` array before entering the loop:

```
var vertices = mesh.vertices;
for(int i = 0; i < vertices.Length; i++)
{
    float x, y, z;
    x = vertices[i].x;
    y = vertices[i].y;
    z = vertices[i].z;
```

```
    // ...

    DoSomething(x, y, z);

}
```

While the CPU cost of accessing a property once is not very high, repeated accesses within tight loops create CPU performance hotspots. Further, repeated accesses unnecessarily expand the managed heap.

This problem is extremely common on mobile, because the `Input.touches` API behaves similarly to the above. It is extremely common for projects to contain code similar to the following, where an allocation occurs each time the `.touches` property is accessed.

```
for ( int i = 0; i < Input.touches.Length; i++ )

{

   Touch touch = Input.touches[i];

    // …

}
```

This can, of course, be trivially improved by hoisting the array allocation out of the loop condition:

```
Touch[] touches = Input.touches;

for ( int i = 0; i < touches.Length; i++ )

{

   Touch touch = touches[i];

   // …

}
```

However, there are now versions of many Unity APIs that do not cause memory allocations. These should generally be favored, when they're available.

```
int touchCount = Input.touchCount;

for ( int i = 0; i < touchCount; i++ )

{

   Touch touch = Input.GetTouch(i);

   // …

}
```

Converting the above example to the allocation-less Touch API is simple:

Note that the property access (`Input.touchCount`) is still kept outside the loop condition in order to save the CPU cost of invoking the property's `get` method.

Empty array reuse

Some teams prefer to return empty arrays instead of `null` when an array-valued method needs to return an empty set. This coding pattern is common in many managed languages, particularly C# and Java.

In general, when returning a zero-length array from a method, it is considerably more efficient to return a pre-allocated singleton instance of the zero-length array than to repeatedly create empty arrays(5) (**Note:** Naturally, an exception should be made when the array is resized after being returned).

Footnotes

(1) This is because, on most platforms, readback from GPU memory is extremely slow. Reading a Texture from GPU memory into a temporary buffer for use by CPU code (e.g. `Texture.GetPixel`) would be very nonperformant.

(2) Strictly speaking, all non-null reference-typed objects and all boxed value-typed objects must be allocated on the managed heap.

(3) The exact timing is platform-dependent.

(4) Note that this is **not** identical to the number of bytes temporarily allocated during a given frame. The profile displays the number of bytes allocated in a specific frame, even if some/all of the allocated memory is reused in

subsequent frames.

(5) Naturally, an exception should be made when the array is resized after being returned.

2018–03–05 Page amended with limited editorial review

Strings and text

Leave feedback

Handling strings and text is a common source of performance problems in Unity projects. In C#, all strings are immutable. Any manipulation of a string results in the allocation of a full new string. This is relatively expensive, and repeated string concatenations can develop into performance problems when performed on large strings, on large datasets, or in tight loops.

Further, as N string concatenations require the allocation of N–1 intermediate strings, serial concatenations can also be a major cause of managed memory pressure.

For cases where strings must be concatenated in tight loops or during each frame, use a StringBuilder to perform the actual concatenation operations. The StringBuilder instance can also be reused to further minimize unnecessary memory allocation.

Microsoft maintains a list of best practices for working with strings in C#, which can be found here on the MSDN website: msdn.microsoft.com.

Locale coercion and ordinal comparisons

One of the core performance problems often found in string-related code is the unintended use of the slow, default string APIs. These APIs were built for business applications, and attempt to deal with handling strings from many different cultural and linguistic rules with regards to the characters found in text.

For example, the following example code returns true when run under the US-English locale, but returns false for many European locales(1)

NOTE: Note that, as of Unity 5.3 and 5.4, Unity's scripting runtimes always run under the US English (en-US) locale:

```
String.Equals("encyclopedia", "encyclopædia");
```

For most Unity projects, this is entirely unnecessary. It is roughly ten times faster to use the ordinal comparison type, which compares strings in a manner familiar to C and C++ programmers: by simply comparing each sequential byte of the string, without regard for the character represented by that byte.

Switching to ordinal string comparison is as simple as supplying `StringComparison.Ordinal` as the final argument to `String.Equals`:

```
myString.Equals(otherString, StringComparison.Ordinal);
```

Inefficient built-in string APIs

Beyond switching to ordinal comparisons, certain C# `String` APIs are known to be extremely inefficient. Among these are `String.Format`, `String.StartsWith` and `String.EndsWith`. `String.Format` is difficult to replace, but the inefficient string comparison methods are trivially optimized away.

While Microsoft's recommendation is to pass `StringComparison.Ordinal` into any string comparison that does not need to be adjusted for localization, Unity benchmarks show that the impact of this is relatively minimal compared to a custom implementation.

Method	Time (ms) for 100k short strings
`String.StartsWith`, default culture	137
`String.EndsWith`, default culture	542

Method	Time (ms) for 100k short strings
`String.StartsWith`, ordinal	115
`String.EndsWith`, ordinal	34
Custom `StartsWith` replacement	4.5
Custom `EndsWith` replacement	4.5

Both `String.StartsWith` and `String.EndsWith` can be replaced with simple hand-coded versions, similar to the example attached below.

```
public static bool CustomEndsWith(string a, string b) {
    int ap = a.Length - 1;
    int bp = b.Length - 1;

    while (ap >= 0 && bp >= 0 && a [ap] == b [bp]) {
        ap--;
        bp--;
    }
    return (bp < 0 && a.Length >= b.Length) ||

            (ap < 0 && b.Length >= a.Length);
    }

public static bool CustomStartsWith(string a, string b) {
    int aLen = a.Length;
    int bLen = b.Length;
    int ap = 0; int bp = 0;

    while (ap < aLen && bp < bLen && a [ap] == b [bp]) {
    ap++;
    bp++;
    }

    return (bp == bLen && aLen >= bLen) ||

            (ap == aLen && bLen >= aLen);
}
```

Regular Expressions

While Regular Expressions are a powerful way to match and manipulate strings, they can be extremely performance-intensive. Further, due to the C# library's implementation of Regular Expressions, even simple boolean `IsMatch` queries allocate large transient datastructures "under the hood." This transient managed memory churn should be deemed unacceptable, except during initialization.

If regular expressions are necessary, it is strongly recommended to not use the static `Regex.Match` or `Regex.Replace` methods, which accept the regular expression as a string parameter. These methods compile the regular expression on-the-fly and do not cache the generated object.

This example code is an innocuous one-liner.

```
Regex.Match(myString, "foo");
```

However, each time it's executed, it generates 5 kilobytes of garbage. A simple refactoring can eliminate much of this garbage:

```
var myRegExp = new Regex("foo");

myRegExp.Match(myString);
```

In this example, each call to `myRegExp.Match` "only" results in 320 bytes of garbage. While this is still expensive for a simple matching operation, it is a considerable improvement over the previous example.

Therefore, if the regular expressions are invariant string literals, it is considerably more efficient to precompile them by passing them as the first parameter of the Regex object's constructor. These precompiled Regexes should then be reused.

XML, JSON and other long-form text parsing

Parsing text is often one of the heaviest operations that occurs at loading time. In some cases, the time spent parsing text can outweigh the time spent loading and instantiating **Assets**.

The reasons behind this depend on the specific parser used. C#'s built-in XML parser is extremely flexible, but as a result, it is not optimizable for specific data layouts.

Many third-party parsers are built on reflection. While reflection is an excellent choice during development (because it allows the parser to rapidly adapt to changing data layouts), it is notoriously slow.

Unity has introduced a partial solution with its built-in JSONUtility API, which provides an interface to Unity's serialization system that reads/emits JSON. In most benchmarks, it is faster than pure C# JSON parsers, but it has the same limitations as other interfaces to Unity's serialization system – it cannot serialized many complex data types, such as Dictionaries, without additional code(2) (NOTE: See the ISerializationCallbackReceiver interface for one way to easily add the additional processing necessary to convert to/from complex data types during Unity's serialization process.).

When encountering performance problems that arise from textual data parsing, consider three alternative resolutions.

Option 1: Parse at build time

The best way to avoid the cost of text parsing is to entirely eliminate the parsing of text at runtime. In general, this means "baking" the textual data into a binary format via some sort of build step.

Most developers who opt for this route move their data to some sort of ScriptableObject-derived class hierarchy and then distribute the data via AssetBundles. For an excellent discussion of using ScriptableObjects, see Richard Fine's Unite 2016 talk on youtube.

This strategy offers the best possible performance, but is only suitable for data that does not need to be generated dynamically. It is best suited for game design parameters and other content.

Option 2: Split and lazy load

A second possibility is to split up the data that must be parsed into smaller chunks. Once split, the cost of parsing the data can be spread across several frames. In the ideal case, identify the specific portions of the data that are required to present the desired

experience to the user and load only those portions.

In a simple example: if the project were a platform game, it would not be necessary to serialize data for all the levels together into one giant blob. If the data were split into individual Assets for each level, and perhaps segmented the levels into regions, the data could be parsed as the player approached it.

While this sounds easy, in practice it requires a substantial investment in tool code and may require data structures to be reorganized.

Option 3: Threads

For data that is parsed entirely into plain C# objects, and does not require any interaction with Unity APIs, it is possible to move the parsing operations to worker threads.

This option can be extremely powerful on platforms with a significant number of cores(3) (NOTE: Note that **iOS** devices have at most 2 cores. Most Android devices have 2–4. This technique of more interest when building for standalone and console build targets.) However, it requires careful programming to avoid creating deadlocks and race conditions.

Projects that choose to implement threading generally use the built-in C# Thread and ThreadPool classes (see msdn.microsoft.com) to manage their worker threads, along with the standard C# synchronization classes.

Footnotes

(1) Note that, as of Unity 5.3 and 5.4, Unity's scripting runtimes always run under the US English (en-US) locale.

(2) See the ISerializationCallbackReceiver interface for one way to easily add the additional processing necessary to convert to/from complex data types during Unity's serialization process.

(3) Note that iOS devices have at most 2 cores. Most Android devices have 2–4. This technique is of more interest when building for standalone and console build targets.

The Resources folder

Leave feedback

The *Resources* folder is a common source of many problems in Unity projects. Improper use of the *Resources* folder can bloat the size of a project's build, lead to uncontrollable excessive memory utilization, and significantly increase application startup times.

These problems are described in detail in the Best Practice Guide AssetBundles and Resources. In particular, refer to the Resources chapter for more information on the above problems.

General Optimizations

Leave feedback

There are as many different ways to optimize code as there are reasons for performance problems. In general, it is strongly recommended that developers closely profile their applications before attempting to apply CPU optimizations. However, there are several simple CPU optimizations that are universally applicable.

Address Properties by ID

Unity does not use string names to address Animator, Material and **Shader** properties internally. For speed, all property names are hashed into property IDs, and it is these IDs that are actually used to address the properties.

Therefore, whenever using a *Set* or *Get* method on an Animator, Material or Shader, use the integer-valued method instead of the string-valued methods. The string methods simply perform string hashing and then forward the hashed ID to the integer-valued methods.

The property IDs created from string hashes are deterministic over the course of a single run. The simplest way to use them is to declare a static read-only integer variable for each property name, and use the integer variable in place of the string. These are automatically initialized during startup with no further initialization code needed.

The appropriate APIs are Animator.StringToHash for Animator property names, and Shader.PropertyToID for Material & Shader property names.

Use non-allocating physics APIs

In Unity 5.3 and onwards, non-allocating versions of all Physics query APIs have been introduced. Replace RaycastAll calls with RaycastNonAlloc, SphereCastAll calls with SphereCastNonAlloc, and so on. For 2D applications, there are also non-allocating versions of all Physics2D query APIs.

Null comparisons against UnityEngine.Object subclasses

The Mono and **IL2CPP** runtimes treat instances of classes that derive from UnityEngine.Object in a specific way. Invoking methods on the instances actually calls into engine code, which must perform lookups and validations to convert the script references to the native references. While small, the cost of comparing a variable of this type to null is much more expensive than a comparison against a purely C# class. For this reason, avoid these null comparisons in tight loops or in code that runs every frame.

Vector and quaternion math and order of operations

For vector and **quaternion** math that is located in tight loops, remember that integer math is faster than floating-point math, and floating-point math is faster than vector, matrix or **quaternion** math.

Therefore, whenever commutative or associative arithmetic allows, attempt to minimize the cost of individual mathematical operations:

```
Vector3 x;

int a, b;
```

```
// Less efficient: results in two vector multiplications

Vector3 slow = a * x * b;

// More efficient: one integer mult, one vector mult

Vector3 fast = a * b * x;
```

Built-In ColorUtility

It is common for applications that must convert between HTML-formatted color strings (`#RRGGBBAA`) and Unity's native `Color` and `Color32` structures to use a script from the Unify Community. This script was both slow and caused extensive memory allocation due to string manipulation.

As of Unity 5, there is a built-in ColorUtility API that performs these conversions efficiently. Usage of the built-in API should be preferred.

Find and FindObjectOfType

It is a general best practice to eliminate all usage of `Object.Find` and `Object.FindObjectOfType` in production code. As these APIs require Unity to iterate over all **GameObjects** and **Components** in memory, they rapidly become non-performant as the scope of a project grows.

An exception to the above rule can be made in accessors for singleton objects. A global manager object often exposes an "instance" property, and often has a `FindObjectOfType` call in the getter to detect pre-existing instances of the singleton:

```
class SomeSingleton {

    private SomeSingleton _instance;

    public SomeSingleton Instance {

        get {

            if(_instance == null) {

                _instance =

                    FindObjectOfType<SomeSingleton>();

            }
```

```
            if(_instnace == null) {

                _instance = CreateSomeSingleton();

            }

            return _instance;

        }

    }

}
```

While this pattern is generally acceptable, it is important to examine the code and ensure that the accessor is be called in **Scenes** where the singleton object does not exist. If the getter does not automatically create an instance of a missing singleton, it is very common to discover that code looking for the singleton results in repeated calls to `FindObjectOfType` (often multiple times per frame) and creates an undesirable drain on performance.

Camera locators

Internally, Unity's `Camera.main` property calls `Object.FindObjectWithTag`, a specialized variant of `Object.FindObject`. Accessing this property is no more efficient than a call to `Object.FindObjectOfType`. If code must address the main **camera**, it is strongly recommended to do one of two things:

Access `Camera.main` in a `Start` or`OnEnable` callback and cache the resulting reference.

Construct a `Camera Manager` class that can provide or inject a reference to the active camera.

Debug code & the [conditional] attribute

The `UnityEngine.Debug` logging APIs are not stripped from non-development builds, and do write to log files if called. As most developers do not intend to write debug information in non-development builds, it is recommended to wrap development-only logging calls in custom methods, like so:

```
    public static class Logger {

        [Conditional("ENABLE_LOGS")]

        public static void Debug(string logMsg) {

            UnityEngine.Debug.Log(logMsg);
```

```
        }

    }
```

By decorating these methods with the [Conditional] attribute, the define or defines used by the Conditional attribute determine whether the decorated method is included in the compiled source.

If none of the defines passed to the Conditional attribute are defined, then the decorated method *and* all calls to the decorated method are compiled out. The effect is identical to what would happen if the method and all calls to the method were wrapped in `#if … #endif` preprocessor blocks.

For more information on the `Conditional` attribute, see the MSDN website: msdn.microsoft.com.

Special optimizations

Leave feedback

While the previous section described optimizations applicable to all projects, this section details optimizations that should *not* be applied prior to gathering profiling data. This may be because the optimizations are labor-intensive to implement, may compromise code cleanliness or maintainability in favor of performance, or may resolve problems that only appear at certain magnitudes of scale.

Multidimensional vs. jagged arrays

As described in this StackOverflow article, it is generally more efficient to iterate over jagged arrays than over multidimensional arrays, as multidimensional arrays require a function call.

NOTES:

These are arrays of arrays, and are declared as `type[x][y]` instead of `type[x,y]`.)

This can be discovered by inspecting the IL generated by accessing a multidimensional array, using ILSpy or similar tools.)

When profiled in Unity 5.3, 100 fully sequential iterations over a three-dimensional 100x100x100 array yielded the following timings, which were averaged over 10 runs of the test:

Array type	Total time (100 iterations)
One-Dimensional Array	660 ms
Jagged Arrays	730 ms
Multidimensional Array	3470 ms

The cost of the additional function call can be seen in the disparity between the cost of accessing multidimensional arrays vs. one-dimensional arrays, and the cost of iterating over a non-compact memory structure can be seen in the difference between accessing jagged arrays vs. one-dimensional arrays.

As demonstrated above, the cost of the additional function call heavily outweighs the cost imposed by using a non-compact memory structure.

For highly performance-sensitive operations, it is recommended to use a one-dimensional array. For all other cases where an array with multiple dimensions is required, use a jagged array. Multidimensional arrays should not be used.

Particle System pooling

When pooling **Particle Systems**, be aware that they consume at least 3500 bytes of memory. Memory consumption increases based on the number of modules activated on the Particle System. This memory is *not* released when **Particle Systems** are deactivated; It is only released when they are destroyed.

As of Unity 5.3, most Particle System settings can now be manipulated at runtime. For projects that must pool a large number of different particle effects, it may be more efficient to extract the configuration parameters of the Particle Systems out onto a data-carrier class or structure.

When a particle effect is needed, a pool of “generic” particle effects can then supply the requisite particle effect object. The configuration data can then be applied to the object to achieve the desired graphical effect.

This is substantially more memory-efficient than attempting to pool all possible variants & configurations of Particle Systems used in a given **Scene**, but requires substantial engineering effort to achieve.

Update managers

Internally, Unity tracks lists of objects interested in its callbacks, such as `Update`, `FixedUpdate` and `LateUpdate`. These are maintained as intrusively-linked lists to ensure that list updates occur in constant time. MonoBehaviours are added to/removed from these lists when they are Enabled or Disabled, respectively.

While it is convenient to simply add the appropriate callbacks to the MonoBehaviours that require them, this becomes increasingly inefficient as the number of callbacks grows. There is a small but significant overhead to invoking managed-code callbacks from native code. This results both in degraded frame times when invoking large numbers of per-frame methods, and in degraded instantiation times when instantiating **Prefabs** that contain large numbers of MonoBehaviours (NOTE: The instantiation cost is due to the performance overhead of invoking Awake and OnEnable callbacks on each **Component** in a prefab.).

When the number of MonoBehaviours with per-frame callbacks grows into the hundreds or thousands, it is advantageous to remove these callbacks and instead have MonoBehaviours (or even standard C# objects) attach to a global manager singleton. The global manager singleton can then distribute `Update`, `LateUpdate` and other callbacks to interested objects. This has the additional benefit of allowing code to smartly unsubscribe from callbacks when they would otherwise no-op, thereby shrinking the sheer number of functions that must be called each frame.

The greatest saving is usually realized by eliminating callbacks which rarely execute. Consider the following pseudo-code:

```
void Update() {
    if(!someVeryRareCondition) { return; }
// … some operation …
}
```

If there are large numbers of MonoBehaviours with Update callbacks similar to the above, then a significant amount of the time consumed running Update callbacks are spent switching between native and managed code domains for MonoBehaviour execution that then exit immediately. If these classes instead subscribed to a global Update Manager only while `someVeryRareCondition` were true, and unsubscribed thereafter, time would be saved both code domain switching and on the evaluation of the rare condition.

Using C# delegates in an update manager

It is tempting to use plain C# delegates to implement these callbacks. However, C#'s delegate implementation is optimized for a low rate of subscription and unsubscription, and for a low number of callbacks. A C# delegate performs a full deep-copy of the callback list each time a callback is added or removed. Large lists of callbacks, or large numbers of callbacks subscribing/unsubscribing during a single frame results in a performance spike in the internal `Delegate.Combine` method.

For cases where adds/removes occur at high frequencies, consider using a data structure designed for fast inserts/removes instead of delegates.

Loading thread control

Unity permits developers to control the priority of background threads that are being used to load data. This is particularly important when trying to stream AssetBundles onto disk in the background.

The priority for the main thread and graphics thread are both `ThreadPriority.Normal` – any threads with higher priority preempt the main/graphics threads and cause framerate hiccups, whereas threads with lower priority do not. If threads have an equivalent priority to the main thread, the CPU attempts to give equal time to the threads, which generally results

in framerate stuttering if multiple background threads are performing heavy operations, such as AssetBundle decompression.

Currently, this priority can be controlled in three places.

First, the default priority for **Asset** loading calls, such as `Resources.LoadAsync` and `AssetBundle.LoadAssetAsync`, is taken from the Application.backgroundLoadingPriority setting. As documented, this call also limits the amount of time that the main thread spends integrating Assets (NOTE: Most types of Unity Assets must be "integrated" onto the Main thread. During integration, the Asset initialization is finalized and certain thread-safe operations are performed. This includes scripting callback invocations, such as Awake callbacks. See the "Resource Management" guide for further details.), in order to limit the impact of Asset loading on frame time.

Second, each asynchronous Asset loading operation, as well as each UnityWebRequest request, returns an `AsyncOperation` object to monitor and manage the operation. This `AsyncOperation` object exposes a priority property that can be used to tweak an individual operation's priority.

Finally, WWW objects, such as those returned from a call to `WWW.LoadFromCacheOrDownload`, expose a threadPriority property. It is important to note that WWW objects do not automatically use the `Application.backgroundLoadingPriority` setting as their default value – WWW objects always default to `ThreadPriority.Normal`.

It's important to note that the under-the-hood systems used to decompress and load data differ between these APIs. `Resources.LoadAsync` and `AssetBundle.LoadAssetAsync` are operated by Unity's internal PreloadManager system, which governs its own loading thread(s) and performs its own rate-limiting. `UnityWebRequest` uses its own dedicated thread pool. `WWW` spawns an entirely new thread each time a request is created.

While all other loading mechanisms have a built-in queuing system, WWW does not. Calling `WWW.LoadFromCacheOrDownload` on a very large number of compressed AssetBundles spawns an equivalent number of threads, which then compete with the main thread for CPU time. This can easily result in frame-rate stuttering.

Therefore, when using WWW to load and decompress AssetBundles, it is considered best practice to set an appropriate value for the `threadPriority` of each WWW object that is created.

Mass object movement & CullingGroups

As mentioned in the section on Transform Manipulation, moving large Transform hierarchies has a relatively high CPU cost due to the propagation of change messages. However, in real development environments, it is often impossible to collapse a hierarchy to a modest number of **GameObjects**.

At the same time, it is good development practice to only run enough behavior to maintain the believability of the game world while eliminating behavior the user will not notice – for example, in a Scene with a large number of characters, it is always more optimal to only run Mesh-skinning and animation-driven Transform movement for characters that are on-screen. There is no reason to waste CPU time calculating purely visual elements of the simulation for characters that are off-screen.

Both of these problems can be neatly addressed with an API first introduced in Unity 5.1: CullingGroups.

Instead of directly manipulating a large group of GameObjects in the scene, change the system to manipulate the Vector3 parameters of a group of BoundingSpheres within a CullingGroup. Each BoundingSphere serves as the authoritative repository for a single game-logical entity's world-space position, and receives callbacks when the entity moves near/within the frustum of the CullingGroup's main **camera**. These callbacks can then be used to activate/deactivate code or components (such as Animators) governing behavior that should only run while the entity is visible.

Reducing method call overhead

C#'s string library provides an excellent case study in the cost of adding additional method calls to simple library code. In the section on the built-in string APIs `String.StartsWith` and `String.EndsWith`, it was mentioned that hand-coded replacements are 10–100 times faster than the built-in methods, even when unwanted locale coercion was suppressed.

The key reason for this performance difference is simply the cost of adding additional method calls to tight inner loops. Each method that is invoked must locate the address of the method in memory and push another frame onto the stack. Neither of these operations are free, but in most code they are sufficiently small to ignore.

However, when running small methods in tight loops, the overhead added by introducing additional method calls can become significant – and even dominant.

Consider the following two simple methods.

Example 1:

```
int Accum { get; set; }
Accum = 0;

for(int i = 0; i < myList.Count; i++) {
    Accum += myList[i];
}
```

Example 2:

```
int accum = 0;
int len = myList.Count;

for(int i = 0; i < len; i++) {
    accum += myList[i];
}
```

Both methods calculate the sum of all integers in a C# generic `List<int>`. The first example is a bit more "modern C#" in that it uses an automatically generated property to hold its data values.

While on the surface these two pieces of code appear equivalent, the difference is notable when the code is analyzed for method calls.

Example 1:

```
int Accum { get; set; }
Accum = 0;
```

```
for(int i = 0;
       i < myList.Count;    // call to List::getCount
       i++) {
    Accum       // call to set_Accum
+=      // call to get_Accum
myList[i];  // call to List::get_Value
}
```

So there are four method calls each time the loop executes:

`myList.Count` invokes the `get` method on the `Count` property
The `get` and `set` methods on the `Accum` property must be called
`get` to retrieve the current value of `Accum` so that it can be passed to the addition operation
`set` to assign the result of the addition operation to `Accum`
The *[]* operator invokes the list's *get_Value* method to retrieve the value of the item at a specific index at the list.
Example 2:

```
int accum = 0;
int len = myList.Count;

for(int i = 0;
    i < len;
    i++) {
    accum += myList[i]; // call to List::get_Value
}
```

In this second example, the call to `get_Value` remains, but all other methods have either been eliminated or no longer execute once per loop iteration.

As `accum` is now a primitive value instead of a property, method calls do not need to be made to set or retrieve its value.

As `myList.Count` is assumed to not vary while the loop is running, its access has been moved outside of the loop's conditional statement, so it is no longer executed at the beginning of each loop iteration.

The timings for the two versions reveal the true benefit of removing 75% of the method call overhead from this specific snippet of code. When run 100,000 times on a modern desktop machine:

Example 1 requires 324 milliseconds to execute
Example 2 requires 128 milliseconds to execute
The primary issue here is that Unity performs very little method inlining, if any. Even under **IL2CPP**, many methods do not currently inline properly. This is especially true of properties. Further, *virtual* and interface methods cannot be inlined at all.

Therefore, a method call declared in the source C# is very likely to end up producing a method call in the final binary application.

Trivial properties

Unity provides many “simple” constants on its data types for the convenience of developers. However, in light of the above, it is important to note that these constants are generally implemented as properties that return constant values.

Vector3.zero’s property body is as follows:

```
get { return new Vector3(0,0,0); }
```

Quaternion.identity is very similar:

```
get { return new Quaternion(0,0,0,1); }
```

While the cost of accessing these properties is usually tiny compared to the actual code surrounding them, they can make a small difference when they are executed thousands of times per frame (or more).

For simple primitive types, use a `const` value instead. `Const` values are inlined at compile time – the reference to the `const` variable is replaced with its value.

Note: Because every reference to a `const` variable is replaced with its value, it is inadvisable to declare long strings or other large data types `const`. This unnecessarily bloats the size of the final binary due to all the duplicated data in the final instruction code.

Wherever `const` isn’t appropriate, make a `static readonly` variable instead. In some projects, even Unity’s built-in trivial properties have been replaced with `static readonly` variables, resulting in small improvements in performance.

Trivial methods

Trivial methods are trickier. It is extremely useful to be able to declare functionality once and reuse it elsewhere. However, in tight inner loops, it may be necessary to depart from good coding practices and instead “manually inline” certain code.

Some methods can be eliminated outright. Consider `Quaternion.Set`, `Transform.Translate` or `Vector3.Scale`. These perform very trivial operations and can be replaced with simple assignment statements.

For more complex methods, weigh the profiling evidence for manual inlining against the long-term cost of maintaining the more-performant code.

Making believable visuals in Unity

Leave feedback

This Best Practice Guide looks at the fundamental steps for setting up a good baseline for believable visuals. The advice in this guide is provided by the Spotlight Team at Unity. The Spotlight Team help developers and content creators achieve the desired look and quality for their projects. They hear stories from across the industry, and can identify the common issues that customers face. Questions that often arise when making a **Scene** are “How do we build to a fairly realistic standard visual?” and “How do we make a **Scene** in Unity that looks believable?”.

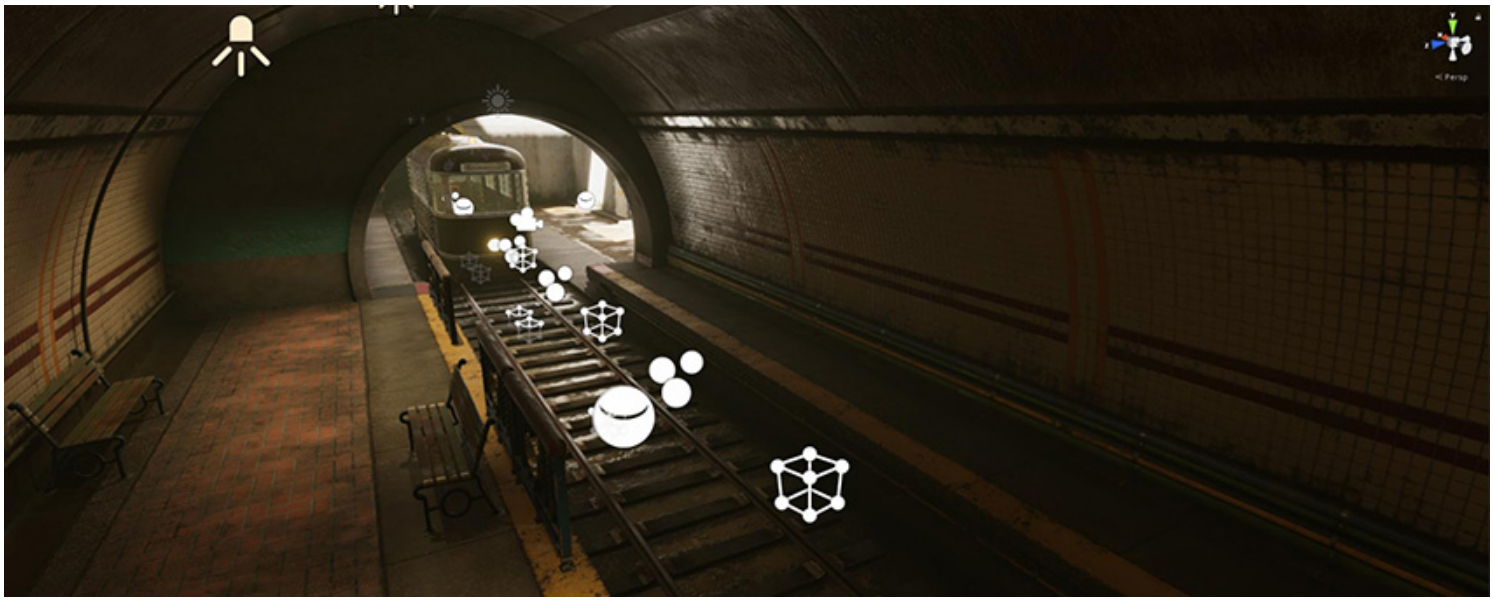
Light, texture, scale, and material all need to work together to make digital content look real. With massive amounts of features, functions, options, properties, systems and variables to tweak, you could ask “Where do I start?” and “In what order do I proceed?”. This guide aims to answer these questions.

Contents

Preparing Assets for Unity
Preparing render settings in Unity
Lighting strategy
Modelling
Shading
Outdoor lighting and Scene setup
Indoor and local lighting
Understanding post-processing
Dynamic lighting

Sample scene

This guide is accompanied by the Spotlight Tunnel sample scene:



To use the sample Scene, extract the Project to a folder and open the Project in Unity.

All **Assets** provided in this sample project should only be used for projects developed with Unity.

2018–10–01 Page published with limited editorial review
Making believable visuals Best Practice Guide added in Unity 2017.3

Preparing Assets for Unity

Leave feedback

The first step is to get your **Assets** into a format suitable for what you want to do. It's very important to set up a proper workflow from your 3D modeling application, such as Autodesk® 3ds Max®, Autodesk® Maya®, Blender, and Houdini, into Unity. When exporting assets from 3D modeling applications for import into Unity, you need to consider:

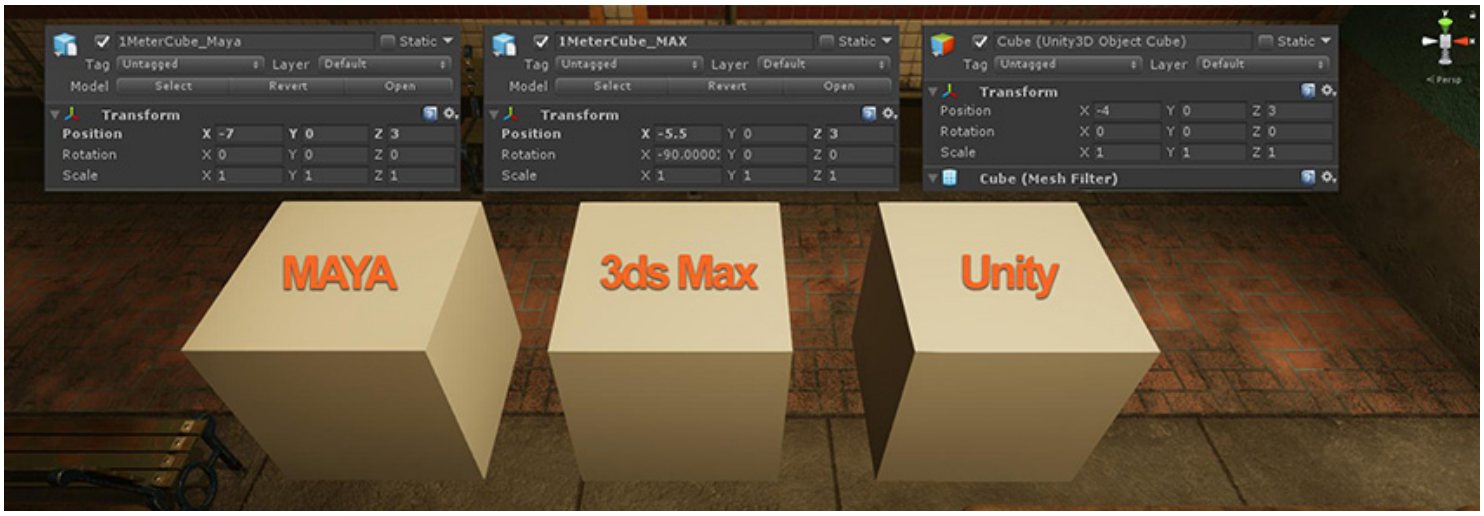
Scale and units
Point of reference scale model
Texture output and channels
Normal map direction

Scale and units

Your project scale, and your preferred unit of measurement, play a very important role in a believable **Scene**. In many "real world" setups, we recommend you assume 1 Unity unit = 1 meter (100cm), because many physics systems assume this unit size. For more advice, see the Art Asset best practice guide.

To maintain consistency between your 3D modeling application and Unity, always validate the imported GameObject scale and size. 3D modeling applications have units and scale settings in the FBX export configuration (see the documentation for your 3D modelling software for configuration advice). Generally, the best way to match the scale when importing to Unity is to set these tools to work in centimeters, and export FBX at automatic scale. However, you should always check that the unit and scale settings match when starting a new project.

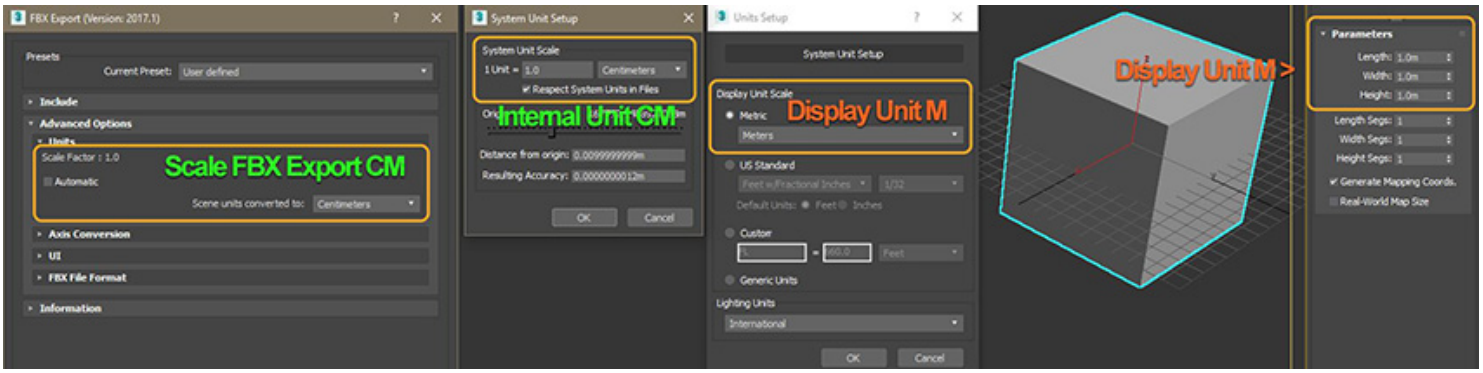
To quickly validate your export settings, In your 3D modeling application, create a simple 1x1x1m cube and import it into Unity. In Unity, create a default Cube (**GameObject** > **3D Object** > **Cube**). This is 1x1x1m. Use this as a scale reference to compare with your imported model. These cubes should look identical when the Transform **component**'s **Scale** property is set to 1,1,1 in the **Inspector**:



Scale comparison using cubes imported from Autodesk® Maya® and Autodesk® 3ds Max®, and a cube created in Unity

Note:

Autodesk® Maya® and Autodesk® 3ds Max® can override your default unit, depending on the last opened file. 3D modeling applications can display different units in the workspace to their internal unit settings, which might cause some confusion.



Differing display and internal units in Autodesk® 3ds Max®

Point of reference scale model

When blocking out a Scene with placeholders or sketching geometry, having a point of reference scale model can be helpful. Choose a point of reference scale model that is appropriate for the Scene you're making. In the Spotlight Tunnel Sample Scene case, we use a park bench:

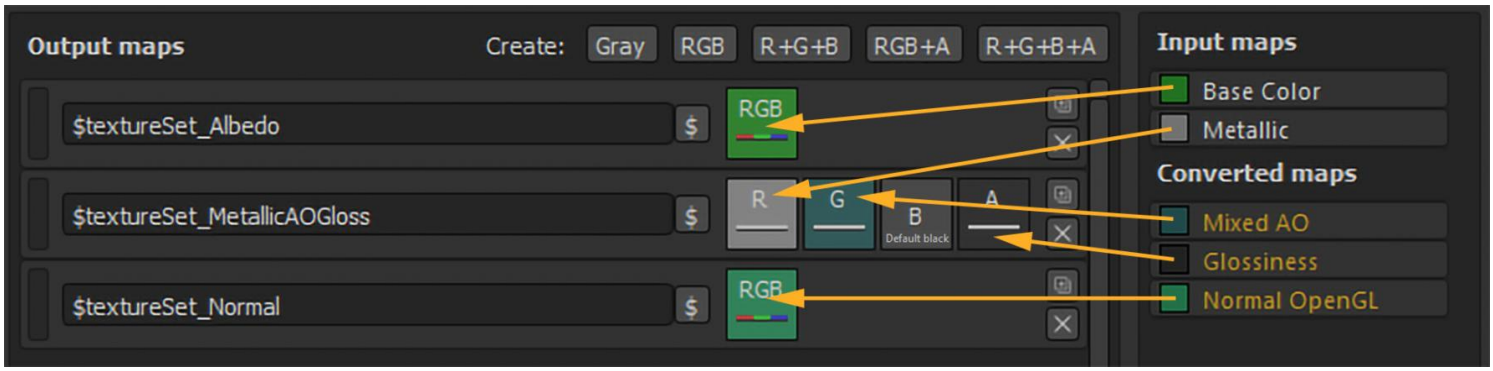
Bench acting as a point of reference scale model

Your Scene doesn't have to use exactly the same proportions as real life. Using a point of reference scale model simply allows consistencies of scale to be relative between GameObjects, even if the Scene is intended to have exaggerated proportions.

Texture output and channels

The information inside a texture needs to contain the correct information to give a proper result when added to a Material. Texture authoring software, such as Photoshop and Substance Painter, outputs consistent and predictable textures when you configure them correctly.

Here's an example of a pre-set configuration for Substance Painter to output textures for use with a Unity Standard Opaque material:



The texture assignment in the Unity Standard Material is:

Output maps for export	Assignment in Unity Standard Shader Material
$textureSet_Albedo	Assigned to Albedo.
$textureSet_MetallicAOGloss	Assigned to Metallic and Occlusion.Smoothness Source set to Metallic Alpha.
$textureSet_Normal	Assigned to Normal Map Slot.

Note: Packing multiple channels to a single texture, such as the MetallicAOGloss, saves texture memory compared to exporting **Ambient Occlusion** (AO) as a separate texture. This is the best way of working with a Unity Standard Material.

When creating textures, it's important not to mix up the alpha channel. The example below shows how transparency in a PNG file can be confusing to author in Photoshop because of the way Photoshop deals with a PNG alpha channel (without using an external plugin). In this case, an uncompressed 32-bit TGA with a dedicated alpha channel might be a better option, assuming the source texture file size is not an issue:

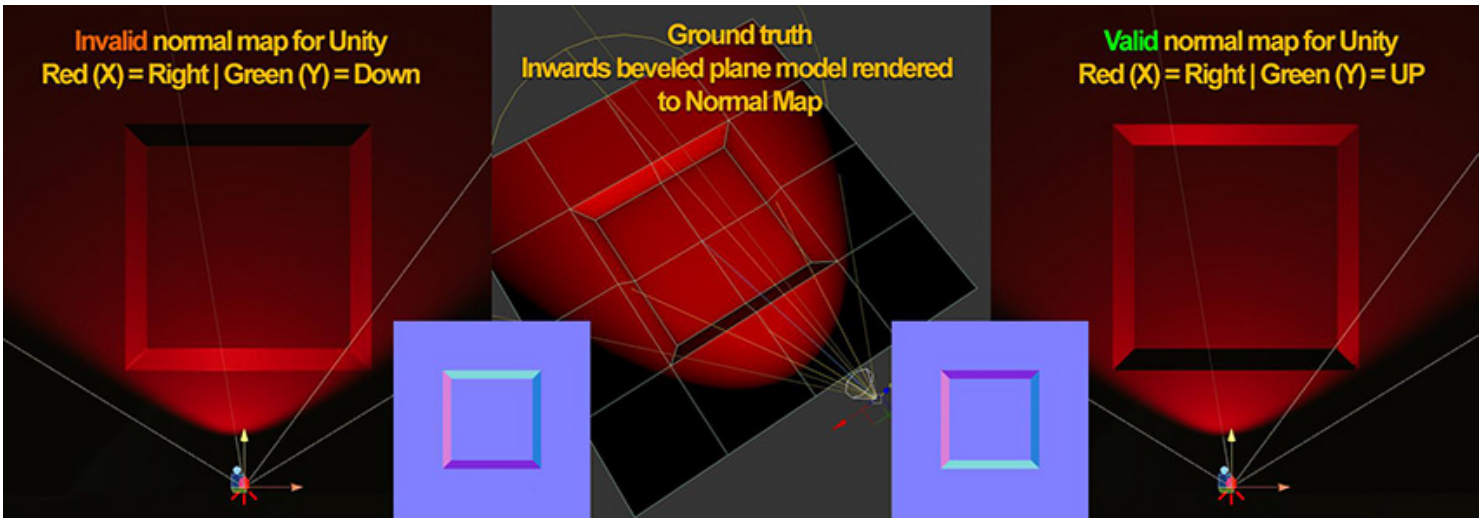
The transparent PNG file shown above was created in Photoshop with its alpha channel coming through as a black value. The TGA with a dedicated alpha channel shows the expected value. As you can see above, when each texture assigned to the Standard **Shader** material reads smoothness data from the alpha channel, the smoothness of the material with PNG textures is unexpectedly inverted, while the smoothness of the material with TGA textures is normal.

Normal map direction

Unity reads tangent space normal maps with the following interpretation:

Red channel X+ as **Right**
Green channel Y+ as **Up**

For example, a Autodesk® 3ds Max® **Render to Texture** normal map outputs the Green Channel Y+ as **Down** by default. This causes an inverted surface direction along the Y axis and creates invalid results when lit. To validate the normal map direction, create a simple plane with concave bevel (middle picture on the example below) and bake it to a flat plane. Then assign the baked normal map into a plane in Unity with identifiable light direction and see if any of the axes are inverted.



Invalid and valid normal map comparison based on Red and Green channel output

For advice on axis settings, refer to your 3D modelling application's documentation.

2018–04–19 Page published with limited editorial review

Making believable visuals Best Practice Guide added in Unity 2017.3

Preparing render settings in Unity

Leave feedback

The following advice helps you understand how to use Unity's rendering features to realistically mimic the real world, and quickly achieve your project's believable visual goal. This covers:

Linear rendering mode
Rendering mode
Enabling post-processing
High Dynamic Range (HDR) Camera
HDR Light map encoding (optional)
Enabling Image Effects for viewport

For more in-depth information, see Unity's lighting and rendering tutorial.

Linear rendering mode

In simple terms, the linear rendering mode prepares Unity to do lighting and shading calculations using physically accurate mathematics before transforming the final output into the format that works best for monitors.

To specify a gamma or linear workflow:

In Unity, select **Edit** > **Project Settings** > **Player** and open **Player Settings**.

Go to **Other Settings** > **Rendering** and change the **Color Space** to **Linear**.

Defining your color space should be one of the earliest decisions in your project because of the drastic impact on the final shading and lighting results. For more advice on each workflow, see Linear or gamma workflow.

Rendering mode

The Spotlight Tunnel Sample **Scene** uses the **deferred shading** rendering path. This allows you to:

Work with multiple dynamic lights efficiently

Combine multiple reflection cubemaps

Use the existing Screen Space Reflection features in Unity 2017+

To set the rendering path in the Editor, select **Edit** > **Project Settings** > **Graphics** or select the **Main Camera** in your Scene, and then set the **Rendering Path** in the Inspector window.

For more information on the rendering mode, see Rendering Pipeline Details.

Enabling post-processing

To display **HDR** lighting properly, install the Unity post-processing stack V1 in your project, and follow these steps to set it up and enable tonemapping:

Create a **Post-Processing Profile Asset** in your project and configure it:

Enable **Color Grading** > **Tonemapper** > **Filmic (ACES)** (Academy Color Encoding Standards). For advice on using ACES tonemapping to handle high intensity color values, such as colored light or fire, see High intensity color.
Enable Dithering. Dithering allows the Scene to alleviate banding artifacts introduced by 8 Bit/channel output from an HDR Scene. Modern engines use this technique to circumvent the limitation of 16M color output.

Leave the rest of the **Tonemapper** settings with their default values.

Select the **Main Camera** in your Scene, click **Add Component** in the **Inspector** window, and choose **Post-Processing Behaviour**.

In the **Profile** field, enter the post-processing profile you created.

The Spotlight Tunnel sample scene uses the post-processing stack V1. For advice on using post-processing stack V2, refer to the package readme.

High Dynamic Range (HDR) Camera

When rendering believable lighting, you're dealing with lighting values and emissive surfaces that have a brightness higher than 1 (high dynamic range), much like real life. You then need to remap these values to the proper screen range (see tonemapping). The high dynamic range setting is crucial because it allows the Unity **camera** to process high values, rather than clip them. To enable HDR, select the **Main Camera** in your Scene and ensure that **Allow HDR** is checked in the Inspector window.

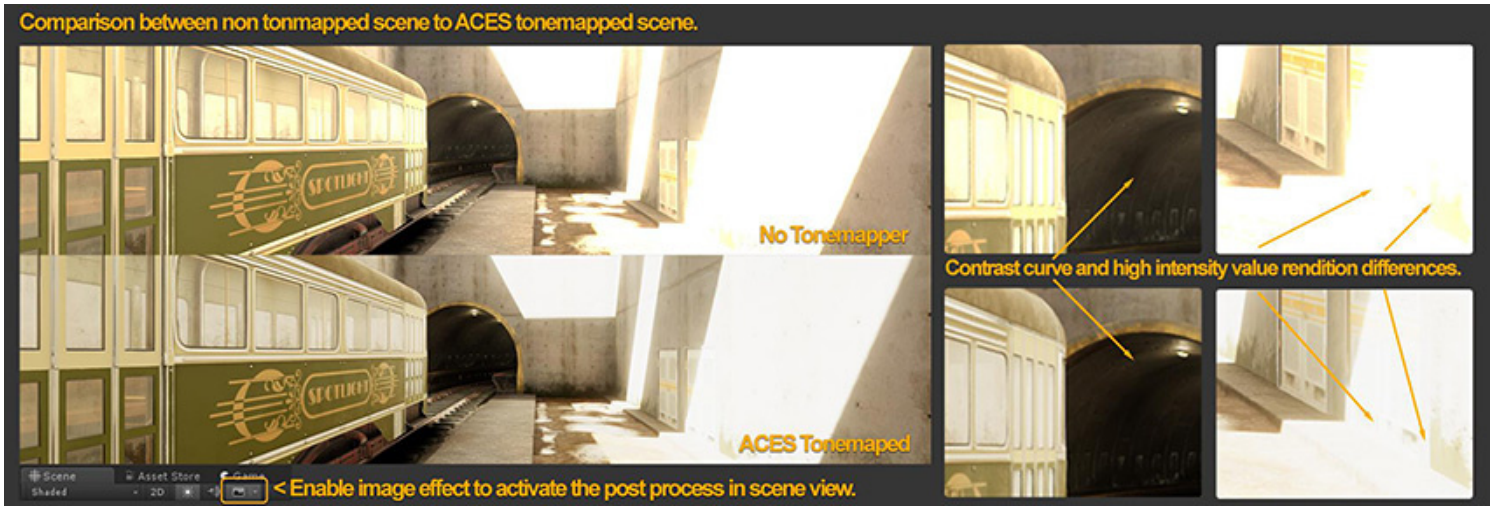
HDR Lightmap encoding (optional)

The Spotlight Tunnel Sample Scene doesn't use baked lighting. However, if you're planning to use HDR baked lighting, set the light map encoding to HDR light map for consistent results.

To set lightmap encoding in Unity, go to **Edit** > **Project** > **Player** > **Other Settingspost** > **Lightmap Encoding**. For more information, see Lightmaps: Technical information.

Enabling image effects in the viewport

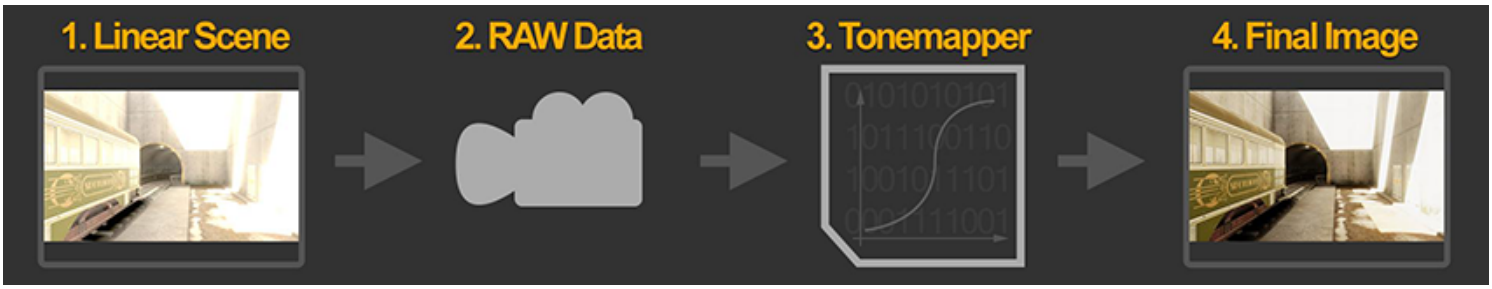
To see the tonemapper while working with the Scene, enable **Image Effects** in the drop-down **toolbar** menu at the top of the **Scene view**. In the following image, notice the highlight rendition and the dark tunnel value separation improvements in the tonemapped Scene. If you look at the non-tonemapped Scene, you can see how the highlights didn't converge to a unified color (the yellowish burning sun in this case).



Comparison between non-tonemapped scene and ACES-tonemapped Scene

This setup essentially tries to replicate how a digital camera captures a Scene with a fixed exposure (without exposure adaptation/eye adaptation features enabled).

At this point, you have a proper foundational Scene rendering setup that should give believable results with a wide range of content:



2018–03–23 Page published with limited editorial review
Making believable visuals Best Practice Guide added in Unity 2017.3

Lighting strategy

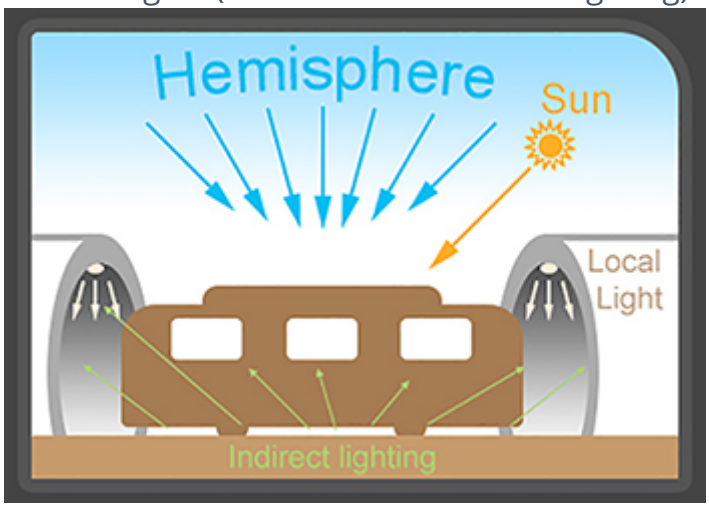
Leave feedback

Before starting to create final **Assets** and approach lighting for a **Scene**, you need to figure out your lighting strategy. Altering your lighting strategy late in development has a high impact on your workflow. Taking the time to get this right before you enter production saves time overall, and allows you to achieve better performance and higher visual fidelity.

As with all development, there's almost always a trade-off between the benefits and impacts of one setup over another. There are times, however, where certain technologies give you options to mitigate these trade-offs within specific constraints. Knowing each feature's trade-offs allows you to choose what's best for your project.

A typical daytime Scene with outdoor areas has the following lighting **components**:

Hemisphere lighting (sky contribution)
Direct lights (Sun + local lights)
Indirect lights (bounced and reflected lighting)



This seems like three simple components. But because you can mix and match **Realtime** lights, **Mixed** lights, **Baked** lights, static **GameObjects** and dynamic **GameObjects**, you can end up with a diverse range of lighting options.

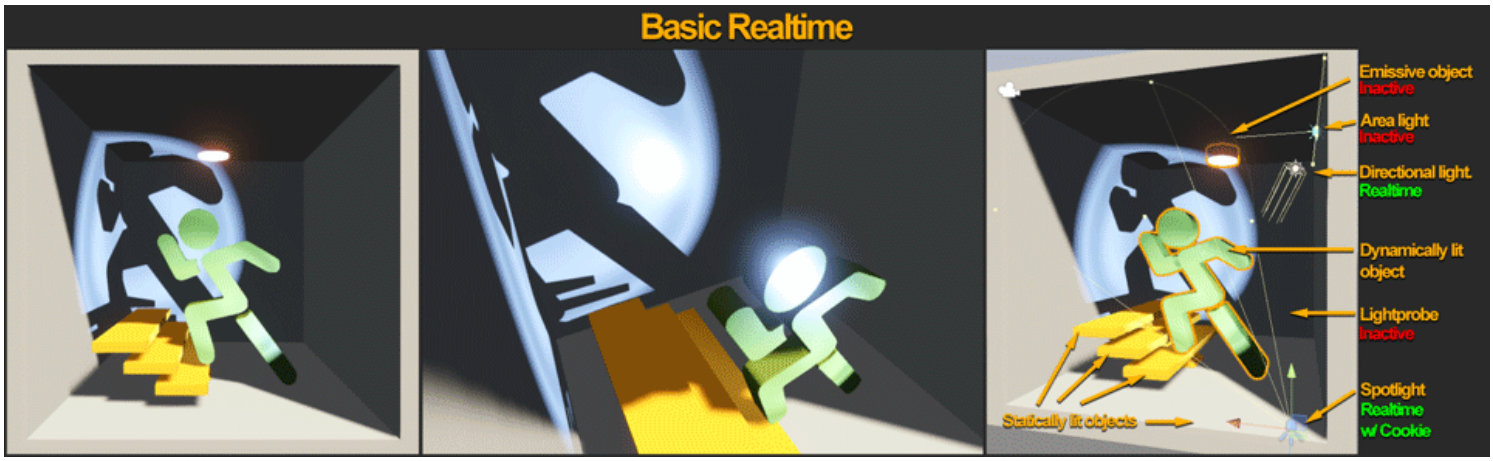
Unity caters to lots of different lighting strategies and project scenarios. To understand lighting modes and setup, see documentation on Lighting Modes.

For newcomers, it can be overwhelming to figure out which setup works best for their Scene, and what the trade-offs are. So let's consider the most commonly used lighting setups:

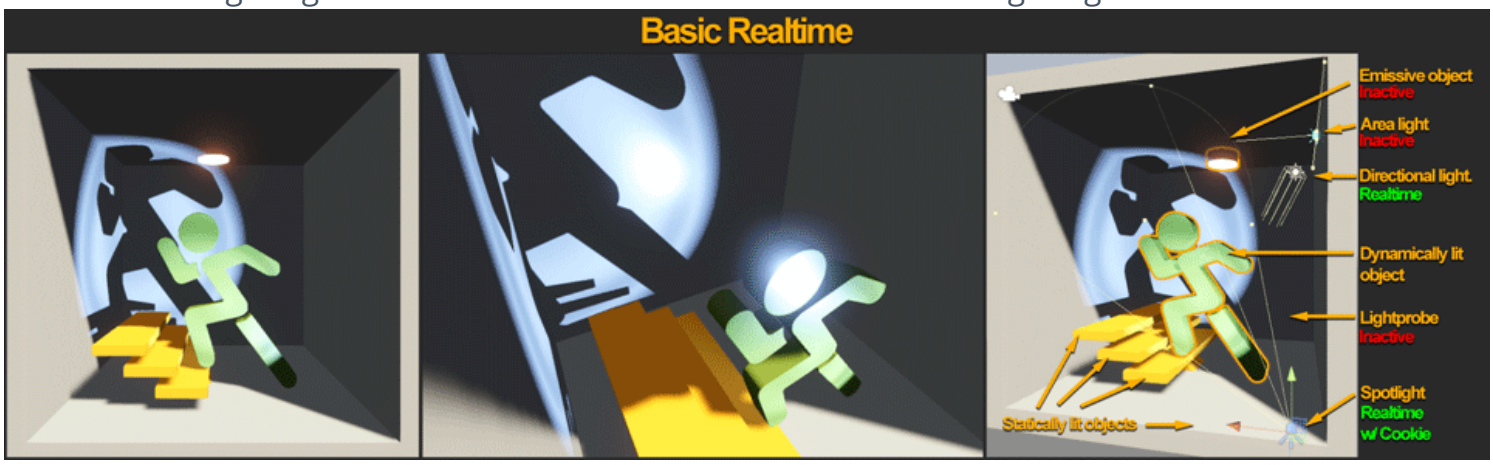
Basic real-time lighting: The specular highlights from the light are visible, but there is no indirect lighting.
Baked lighting: Soft baked shadows are visible, and static indirect lighting is visible in high resolution, but there are no specular highlights from lights, and dynamically lit GameObjects don't cast shadows.
Mixed lighting: Similar to **Baked** lighting, but there is specular response from lights, and dynamically lit GameObjects do cast shadows.
Real-time lighting and GI: Proper indirect lighting response and specular response are visible, lights are all moveable and updateable, but there's no angular soft shadow.

All options enabled): Depending on the settings of each light, you can achieve the combination of all the above options.

The following images show the notable differences between these setups:



Five different lighting modes without Ambient Occlusion or Baked lighting enabled

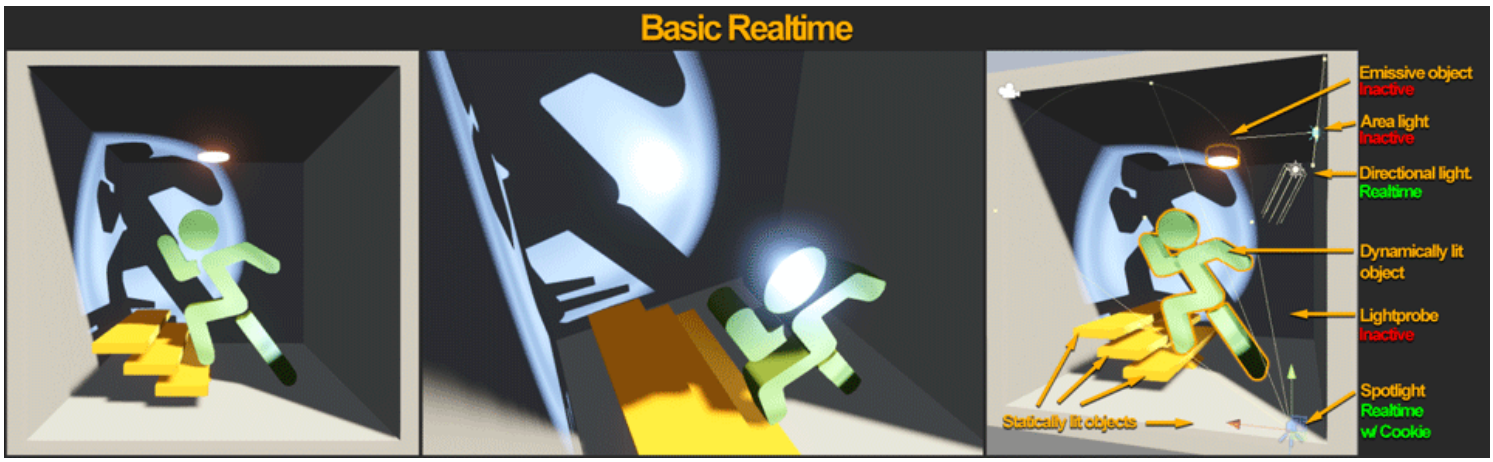


Four different lighting modes with Ambient Occlusion and Baked lighting enabled

Note: **Realtime GI** can't bake static **ambient occlusion**, so it's not included.

Basic real-time lighting and ambient (with no Realtime GI or Baked GI)

Basic real-time lighting is generally used in stylistic visual projects and prototype phases.



Basic real-time lighting

Typical platform target: Console and PC

Advantages:

All direct lights and shadows are applied in real time, and are therefore movable.
Allows for fast iteration, because there is no precomputation, baking, or **mesh** preparation.
Dynamic and Static GameObjects are lit using the same method, so **Light Probes** are not required.
Disadvantages:

No hemisphere occlusion, just **skybox**/ambient value, and the color in the area is not lit by direct lighting.
Without GI or indirect lighting component, the Scene might not give the best visual outcome.

All baked lighting and Light Probes

Baked lighting is generally useful for games where run-time performance is an issue but there's room in memory, such as top-down isometric mobile games and high frame-rate VR games.

Comparison for full baked lighting with and without Ambient Occlusion
Typical platform target: Mobile platform, VR, console and low end PC

Advantages:

All lights are baked for static GameObjects. They produce ambient occlusion and indirect lighting.
Unity can bake area light bake support and soft shadow angles onto statically lit GameObjects.
Fastest run-time performance among the common setups listed here.
Disadvantages:

Can slow down lighting iteration because lights are baked, which means Unity needs to re-compute lights whenever the Scene changes (unless you're using Progressive Lightmapper)
Dynamically lit GameObjects are only lit using Light Probes.
Specular highlights only rely on **cubemaps** and reflections, not on light sources.
There is no shadowing from dynamic GameObjects.
Can require a lot of run-time memory, depending on how many light map textures are used in the Scene.
Might require authoring texture coordinates channel 2 (UV2 for light map) if GameObjects texture charts overlap.

Mixed lighting with Shadowmask and Light Probes

Mixed lighting is often useful in games where time-of-day lighting (such as sun movement) is not important.

Comparison for **Shadowmask Mixed** Lighting with and without Ambient Occlusion
Typical platform target: VR, console and PC

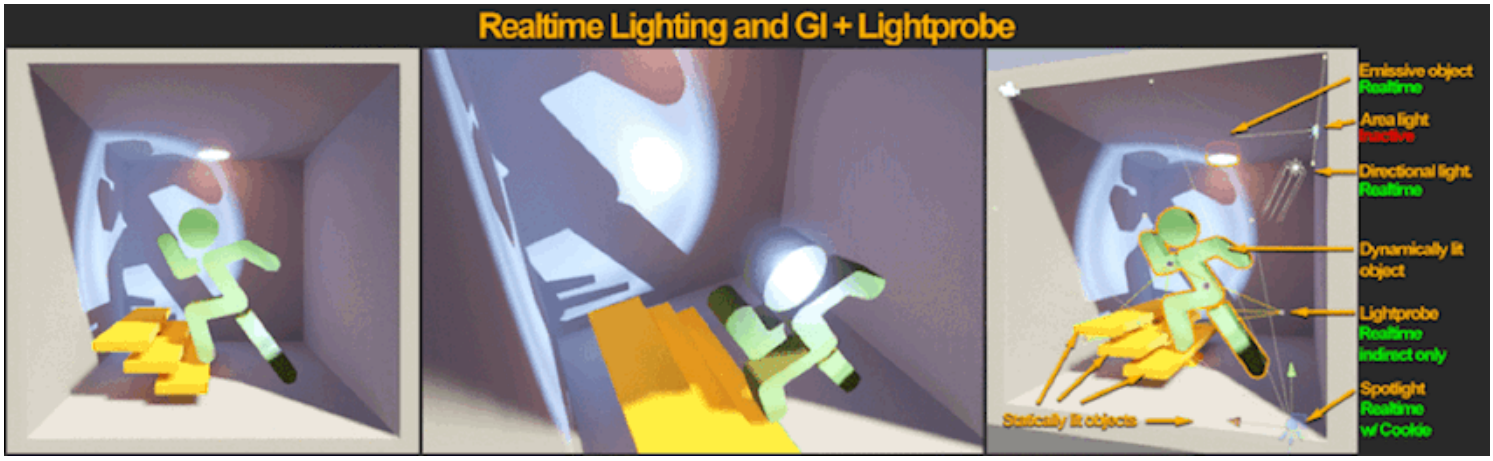
Advantages:

Similar to all Baked lighting, but in Mixed lighting, Dynamic GameObjects get real-time specular lighting and cast real-time shadows, while static GameObjects get baked shadowmasking, resulting in better visual quality.
Disadvantages:

GameObjects have a limit of 4 Shadowmasks. Additional shadow casting lights gets baked.
Rendering real-time lights at run time is more resource-intensive.
Mixed lights can drastically affect performance in certain setups.
For more information on Shadowmask lighting, see Shadowmask.

Real-time lighting with Realtime GI and Light Probes

This setup is useful in open area games where you need time-of-day lighting updates (such as the sun moving) and dynamic lighting effects.



Realtime GI, showcasing indirect lighting updates as the directional light changes
Typical platform target: Console and PC

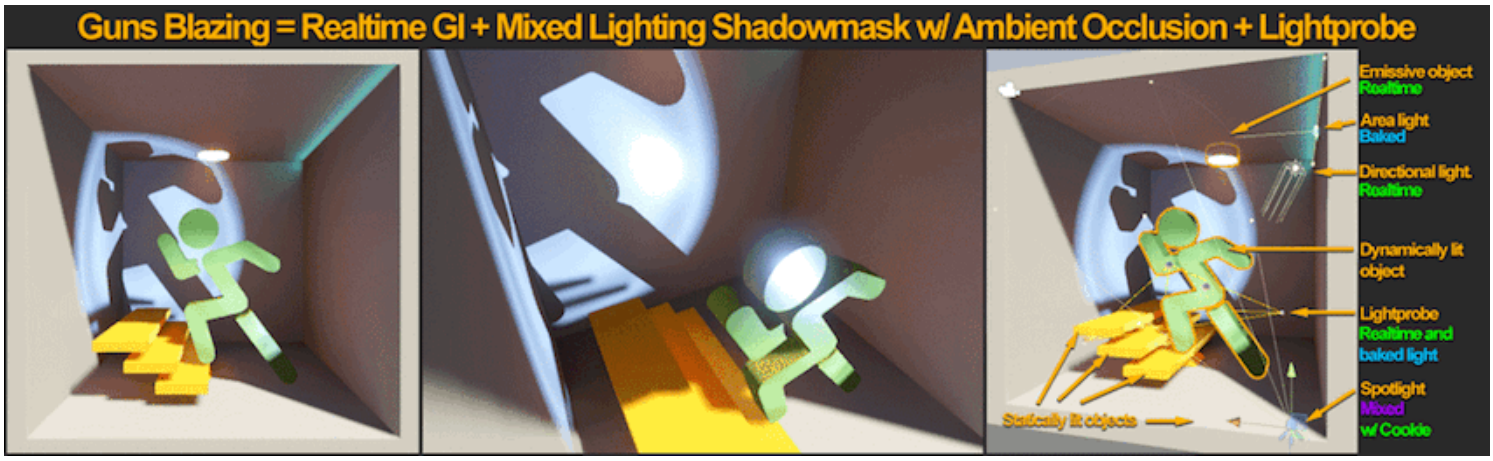
Advantages:

This allows for fast lighting iteration with real-time indirect lighting.
Dynamic and static GameObjects get real-time specular lighting and shadows.
Can use less memory than Baked lighting for indirect lighting.
Has a fixed CPU performance impact for updating global illumination.
Disadvantages:

Occlusion isn't as detailed as Baked lighting, and usually must be augmented by Screen Space Ambient Occlusion (SSAO) and per-object texture baked AO.
No area/light angle soft shadows for static GameObjects.
Real-time lights can drastically affect performance in certain setups.
Precompute times can take a significant amount of time if there are too many GameObjects contributing to the static lighting, especially without an optimized UV setup. For more information, see Global Illumination UVs.
For in-depth information on optimizing **Realtime GI**, see Unity's tutorial on Introduction to Precomputed Realtime GI.

All options enabled

You would only want to generally enable all lighting options in games with high fidelity requirements that have tightly controlled memory usage and performance limits. You should only do this if you fully understand each individual system, and know how to handle each lighting combination.



All options enabled, allowing all possible lighting techniques available in Unity
Typical platform target: Console and PC

Advantages:

This is the complete set of lighting features, giving you full functionality.
Disadvantages:

Has high performance requirements at run time, with high memory usage.
Increases the workflow time, by requiring more UV authoring and baking time.
To help you learn about lighting, the Spotlight Tunnel Sample Scene uses real-time lighting with Realtime GI. This provides a diverse range of specular responses, good bounce lighting, and allows you to quickly iterate on lighting.

2018–03–21 Page published with limited editorial review

Making believable visuals Best Practice Guide added in Unity 2017.3

Modelling

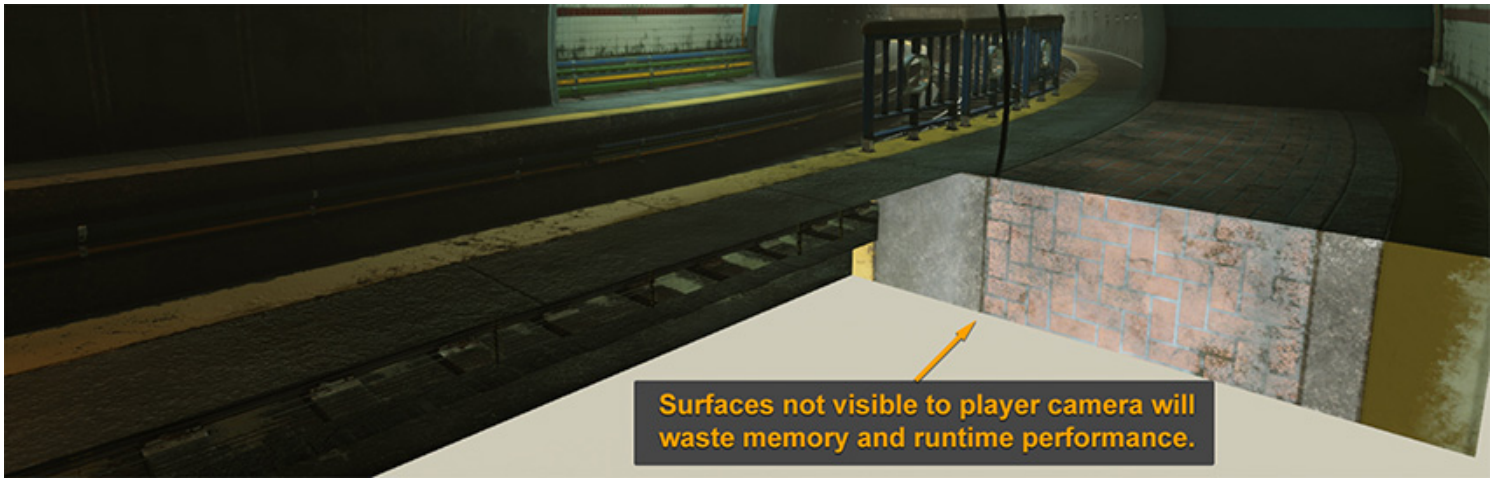
Leave feedback

It's really important to plan ahead before modeling. Quickly made models are fine during pre-production or for roughing out a space, but the moment you start finalizing an **Asset** to make a proper **Scene**, consider the following:

Make every polygon count
GameObject contribution to lighting
Model UV layout strategy
Details in geometry
Smoothing groups (Hard/soft edges of polygons)

Make every polygon count

Despite modern hardware being more capable than ever, simple geometry is always beneficial in a Scene. Unnecessary tessellation and complex geometry is difficult to manage for a real-time setup, and it can impact performance and use memory unnecessarily. The following example demonstrates how geometry that is never seen by the players wastes resources such as **lightmap** and overdraw, and causes light leakage:



Scene highlighting unnecessary visible surfaces

GameObject contribution to lighting

If you are using Baked lighting or Realtime GI with **Light Probes**, you need to decide whether a **GameObject** contributes to lighting in a Scene, or only receives indirect/baked lighting in the Scene.

GameObjects that contribute to lighting

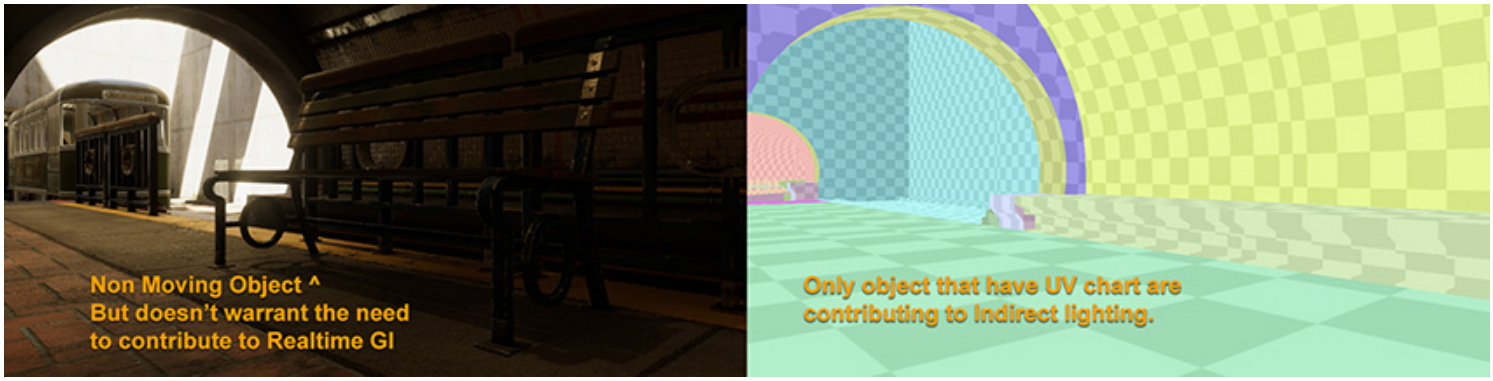
To set a GameObject to contribute to lighting, make sure **Lightmap Static** in checked in the **Inspector** window. This provides simpler and smoother surface areas that produce better indirect bounces/baked lighting because of their efficiency in space usage for lightmap textures. Note that:

You might need to author UV2 for the geometry when doing a light bake, if the auto lightmap UV provides inefficient chart or generate undesirable seams.
You might need to author UV3 for the geometry for efficient results in Realtime GI.
Sometimes, in Realtime GI, you can simplify a UV of a **Mesh** to make the geometry use significantly less resources, and produce the best result with fewer artifacts.

GameObjects that only receive lighting in the scene

When GameObjects only receive lighting from **real-time lights** and Light Probes, the geometry doesn't have a lightmap UV restriction. The geometry still needs special attention if it's large, because it might not be lit properly with a single Light Probe, and might require a Light Probe Proxy Volume component to stitch together multiple probe light definitions.

You don't always need to use a lightmap or use Realtime GI for non-moving GameObjects. If a GameObject is small, or it doesn't have surfaces that bounce much light, it probably doesn't need to be included in the lightmap. The bench and railings shown below are a good example:



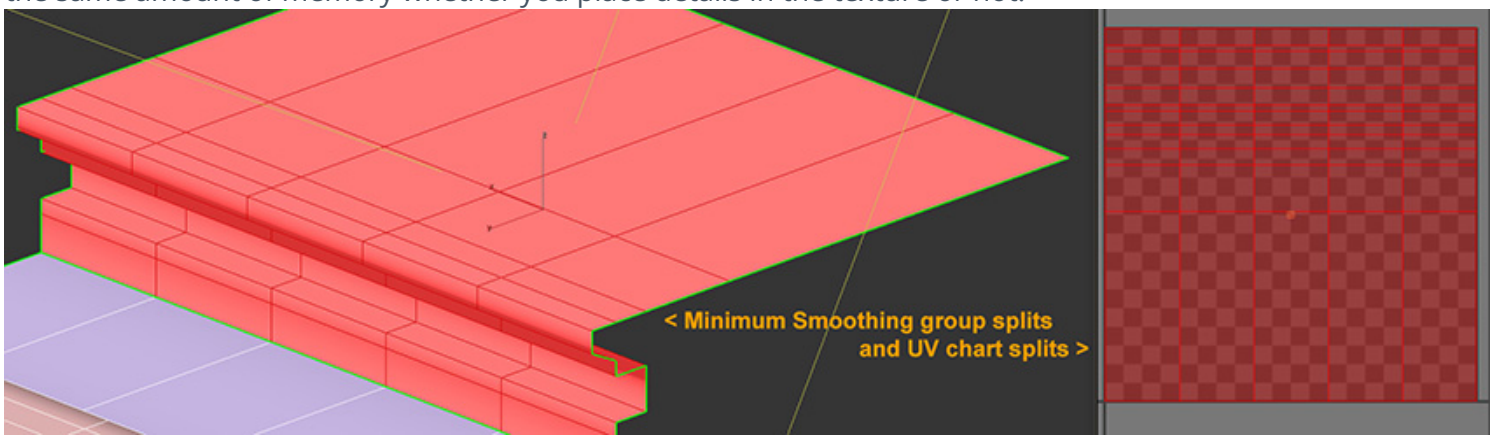
Only GameObjects with a UV chart need to contribute to indirect lighting

Model UV layout strategy

The UV layout can help improve visual quality while using the same amount of memory for normal map baking (typically UV1), lightmaps baking (UV2) and real-time lightmaps (UV3), especially for geometry with non-tileable textures.

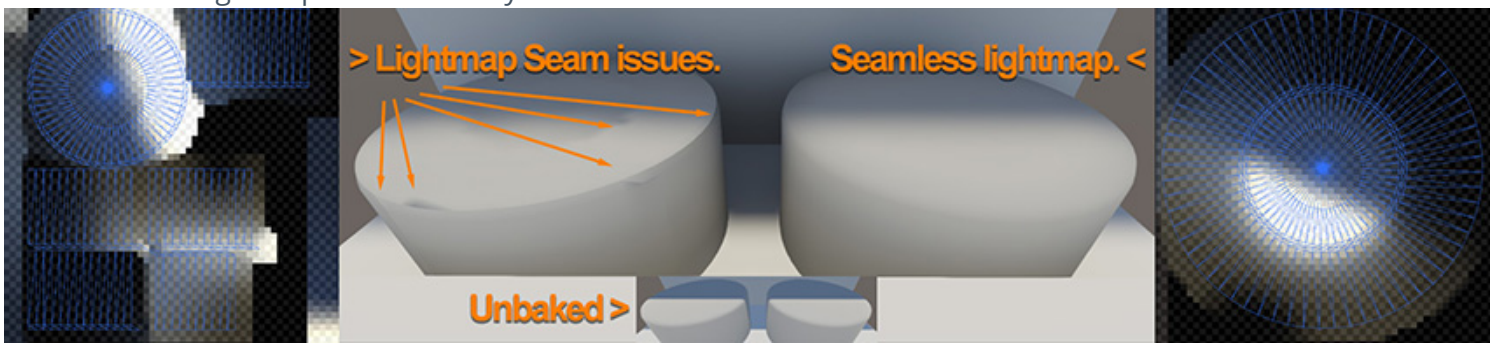
Here are a few tips to consider when making a UV layout strategy:

For UV1 charts, split the UV shell only as necessary, and try to lay out the UV chart as efficiently as possible to prevent wasting texture space for normal map baking. To put it into perspective, a 1024 square texture uses the same amount of memory whether you place details in the texture or not.



Example of how the pieces occupy the whole texture space, avoiding wasted space

For lightmaps (UV2), try to make unbroken lightmap charts with borders that touch but do not overlap, to avoid bleeding or seams in the light bake. Keeping a consistent scale between UV charts/shells is important for even distribution of lightmap texels across your model.



An exaggerated lightmap splitting on a simple geometry, to demonstrate issues with lightmap seams

For Realtime GI (UV3), prioritize UV space for large areas that represent big surfaces in your model to reduce memory usage and avoid seams. In many cases, the automatic UV settings in the model can really help optimize the chart. For in-depth information on chart optimization for Realtime GI, see the Unity tutorial Optimizing Unity's auto unwrapping.

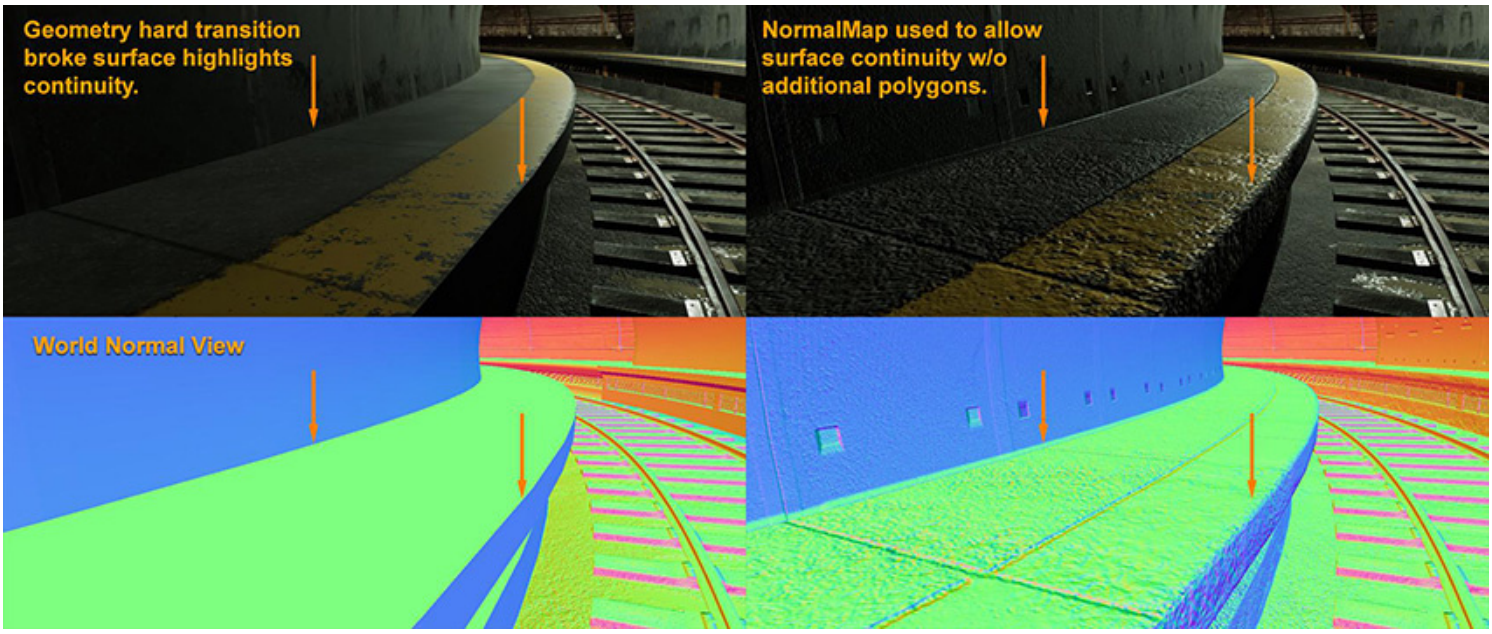
For GameObjects that don't require lightmaps, don't waste memory and time by authoring additional UVs, unless custom **shaders** require them.

Details in geometry

Real world GameObjects are highly detailed. To author real-time geometry, you need to identify which details to place in geometry, and which to place in the normal map and textures. When developing Assets for real-time Scenes, it's normal to bake high-polygon to low-polygon normal maps.

Example of baked tangent space normal map from High Poly mesh to Low Poly mesh

One important detail to remember is the way edges on a GameObject catch highlights. It is unusual to find a real-life object with very sharp edges, with non bevelled edges, or without detailed edge definition. Replicating realistic edge effects improves the believability of the Scene.

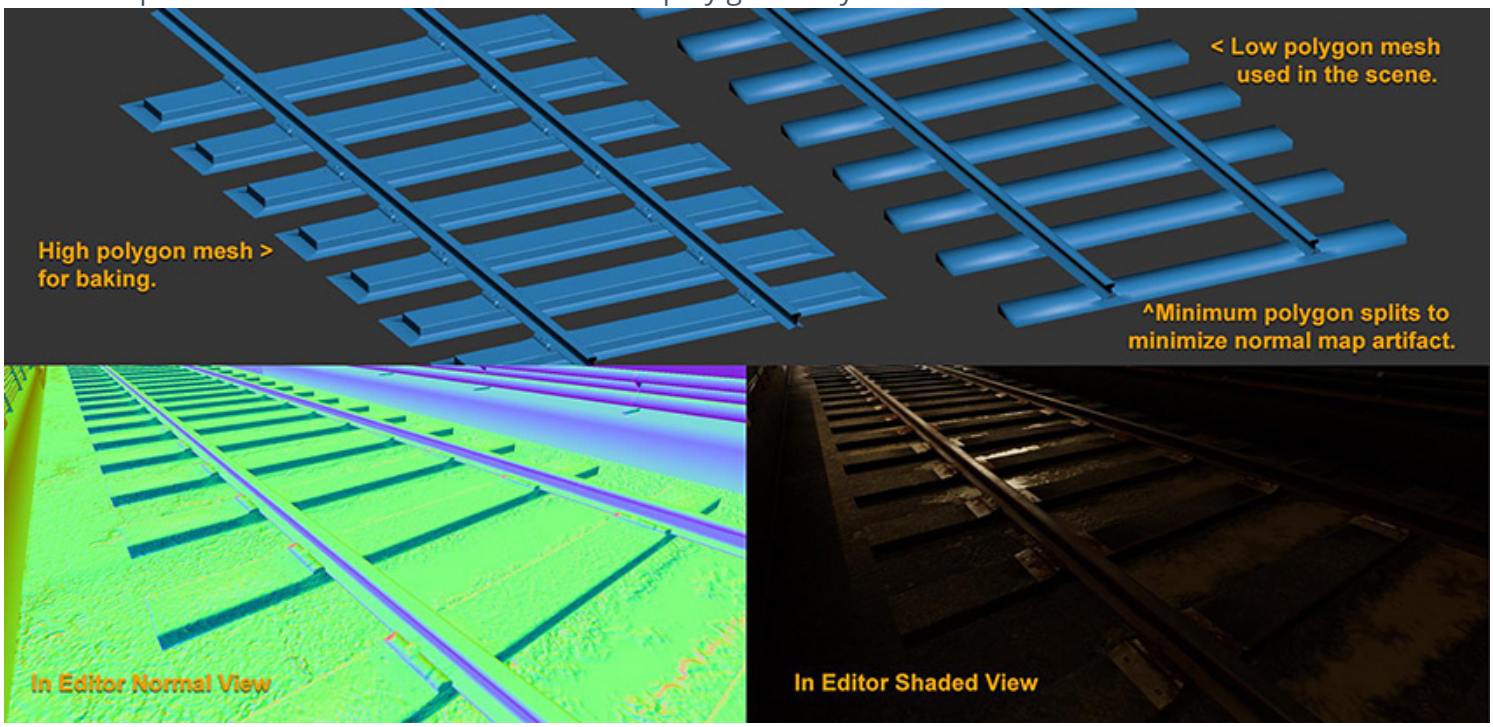


Normal map gives a smooth flowing highlight on surfaces where it meets other geometry

Smoothing groups (Hard/Soft edges of polygon)

You can improve the efficiency of models and normal maps by using proper smoothing groups. Here are some tips on using smoothing groups:

When dealing with normal map baking from high polygon to low polygon, use a simple configuration for smoothing groups rather than multiple faceted polygons. This is because tangent normal maps need to bend the hard split of the surface normal from the low-poly geometry.



A normal map bending the low-poly normal smoothly to mimic the high polygon mesh

A smooth polygon with a good normal map saves on vertex count, which equates to more efficient geometry to render. Here's a simple example that compares the following setups:

A single plane with split smoothing groups that is equal to 36 vertices.

A simple 18 triangle plane in 1 smoothing group that is equal to 16 vertices.

Single plane and triangle plane vertex count comparison

A smooth polygon saves on chart splitting in Lightmap baking and Realtime GI that produces smoother visual result.

2018–03–21 Page published with limited editorial review

Making believable visuals Best Practice Guide added in Unity 2017.3

Shading

Leave feedback

Materials define how light reacts with the surface of a model, and are an essential ingredient in making believable visuals. When you've created a model, you use a **shader** to define its surface properties. The Unity Standard Shader is a built-in shader with a comprehensive set of features, which allow you to create plausible materials easily, and incorporates an advanced lighting model called **Physically Based Shading** (PBS). You should master the use of the Standard Shader before making custom **surface shaders** in **ShaderLab**, ShaderGraph (Scriptable Render Pipeline) or other Shader creation tools.

Most real-life surfaces are made up of multiple materials, so when texturing a **GameObject** with the Unity Standard Shader, you need to consider:

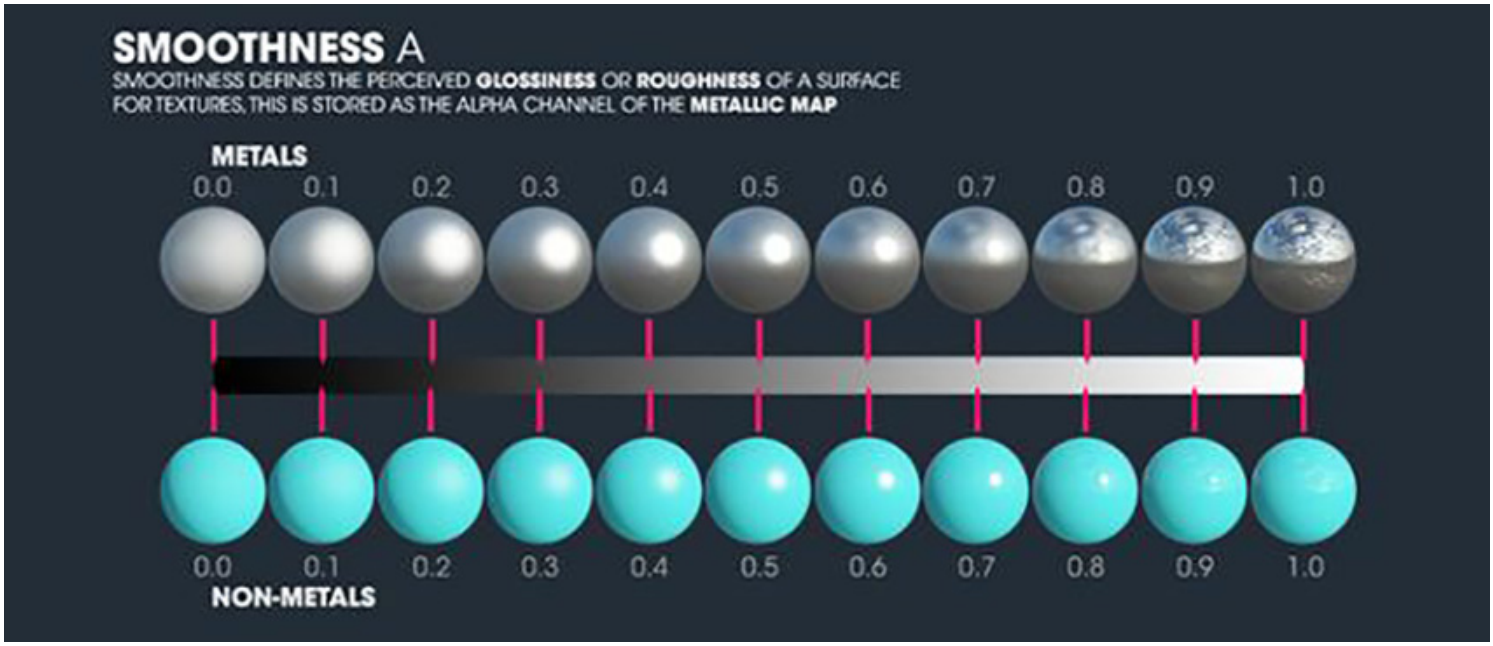
Standard or Standard Specular Setup
Albedo values and Material Validator
Metallic values
Smoothness value
Normal map
Occlusion map
Reference picture, color chart and photo source

Note: Only Albedo, Smoothness, Normal Map and AO covered here. For further advice, see documentation on Standard Shader material parameters.

Standard or Standard Specular setup

In Unity, you can set up a Material as being **Standard** or **Standard Specular**. When working with either of these, be aware that it's generally easier to use a Standard Material, because the Specular brightness and color are calculated automatically based on Albedo, Smoothness and Metallic input.

In Standard setup material, setting Metallic at 1 means the albedo drives the color of the specular and its brightness in tandem with smoothness, which adjusts the brightness and glossiness of the surface.



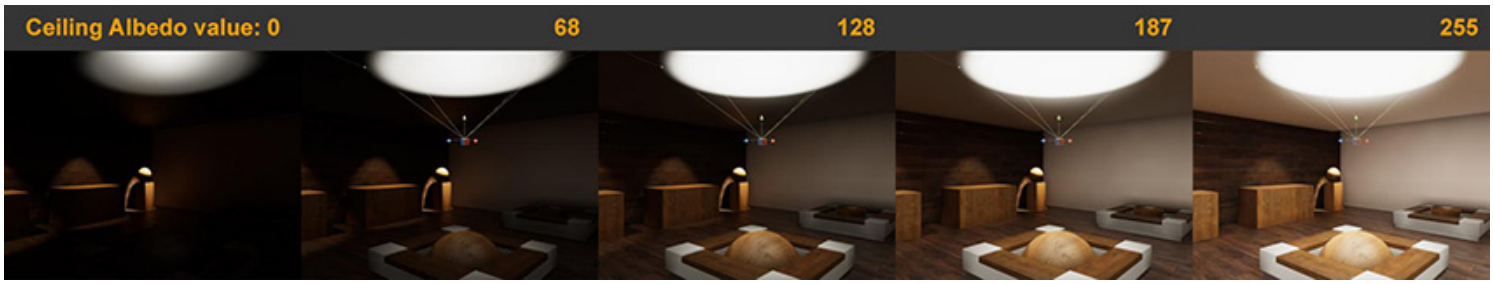
When Metallic is set to 0, the albedo color doesn't affect the specular color, and shows up as the surface color.

Use Standard Specular when you want to un-tether the specular color from the material's albedo. This is the case with non-metal materials or materials that aren't standard metals, where it's better to control the specular color manually. For more information, see Specular mode: Specular parameter.

Albedo values and Material Validator

While a Physically Based Shader works hard to automatically calculate specular brightness and distribution from the light, the albedo of your material needs to be plausible. A material's albedo affects both direct and indirect lighting, and propagates an unrealistic value through the rest of your **Scene**'s lighting.

A very dark albedo significantly absorbs light, and causes an unusual lighting response. An albedo that is too bright reflects significant amounts of light and indirect colors that do not occur in real life.



Albedo set on a non-metal surface affecting indirect lighting

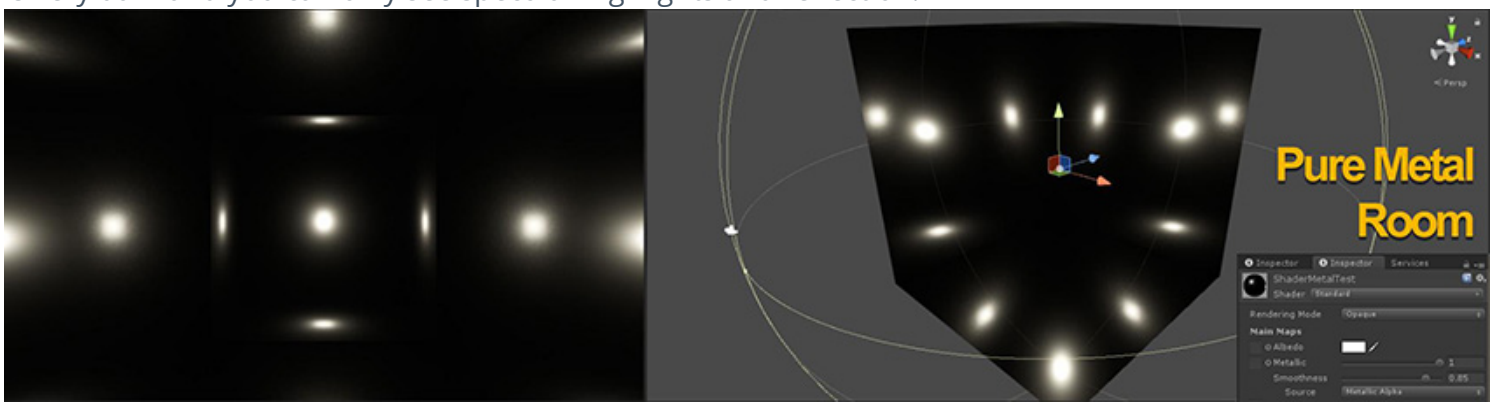
There's no defined value for non-metal painted surfaces, which are very common in real life. For example, you could choose to paint a wooden wall with charcoal black or snow white, but there's no single definitive value of that wall albedo. To help you decide on albedo values, there is a chart of PBS values (see Material charts). As a general guideline, for a non-metal painted surface, an albedo value below 0.2 is too dark and an albedo value above 0.8 is too bright. For darker dielectric material information, see Materials authoring guidelines 1 : dark dielectric materials (PDF)

The Material chart is fairly simple to use when dealing with a single albedo surface. However, deciding on complex albedo textures can be difficult. The Material Validation tool allows you to check whether material values follow the guidelines or not. You can enable this tool in the Scene view by selecting **Material Validation** from the drop down menu in the top left corner of the viewport. **Note**: Material Validation is only available when you're not using the Scriptable Render Pipeline (SRP). To disable SRP, go to **Edit** > **Project Settings** > **Graphics** and set **Scriptable Render Pipeline Settings** to **None**.

Metallic values

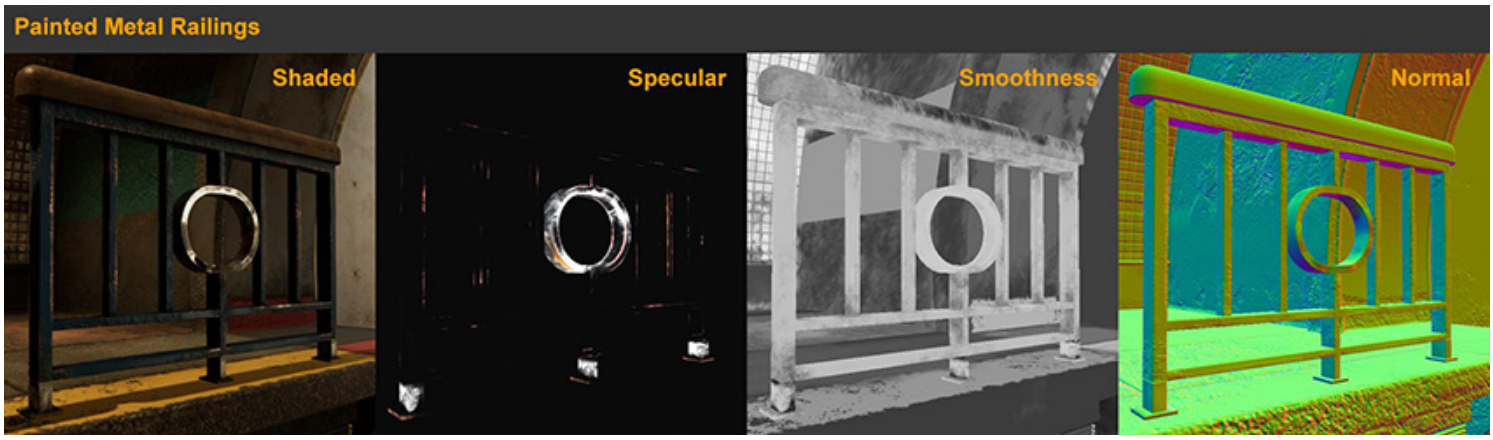
The Metal value of a material defines how much of the environment gets reflected to the surfaces, while also determining how much of the albedo color is visible on the surface. When a surface is pure metal, the surface color (albedo) drives the color of the environment reflection. When working with metal materials, consider the following:

Pure metal gloss materials don't bounce diffuse lighting. If your entire room is made out of metal, your room is very dark and you can only see specular highlights and reflection.



Example of how dark a smooth metal room with full pointlight coverage is

Deciding whether a surface is a metal or not can be difficult. Instead of considering the GameObject's core material, focus on the final surface of a GameObject. For example, metal railings that are painted blue should only have their unpainted area designated as metal.



An example of how you should texture a painted metal railing
Note: While the chipped area of the painted metal bar is metallic, rust is not metal on a surface.

While it is easy to imagine that a material only needs either a metal value of 0 or 1, there are cases where surface materials are mixed in, or blended. For example, metal GameObjects partially covered with dust or dirt have a metal value between 0 and 1 due to blending. Other than that, try to avoid using a metal value between 0 and 1 when creating plausible materials.
For more information about metal, see Metallic mode: Metallic Parameter.

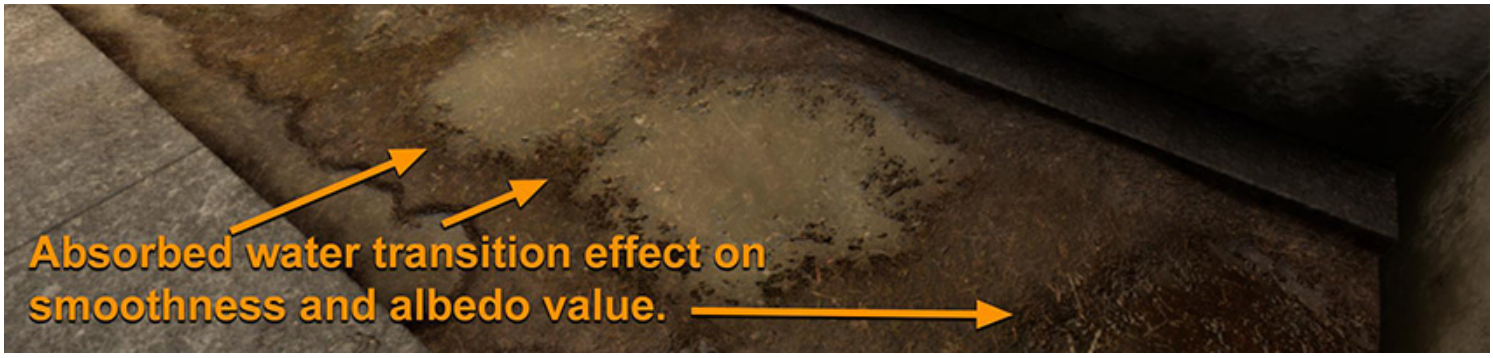
Smoothness value

Smoothness controls the microsurface detail of the surface. A value of 0 yields a very rough, dull surface, and a value of 1 yields a pure reflective mirror-like surface. This is often straightforward and intuitive, but it is useful to consider the following:

Focus on the final surface quality of the GameObject. For example, an object made of concrete could have a rough surface or have a gloss paint on top of it. The smoothness value of unpainted wood depends on whether the wood is polished or not.

Don't forget about scuffs, dirt, scratches and water stains. In real life, the surface of a material is affected by many variables. Materials rarely have a pure single surface.

To determine the characteristic of the material, consider how elements are blended between surfaces. For example, a water puddle on soil usually has a ring of absorbed water that has darkened the albedo, instead of just a direct smoothness blend.



Example of darkened smoothness and albedo on an absorbed water transition
For more information on smoothness, see Smoothness.

Normal map

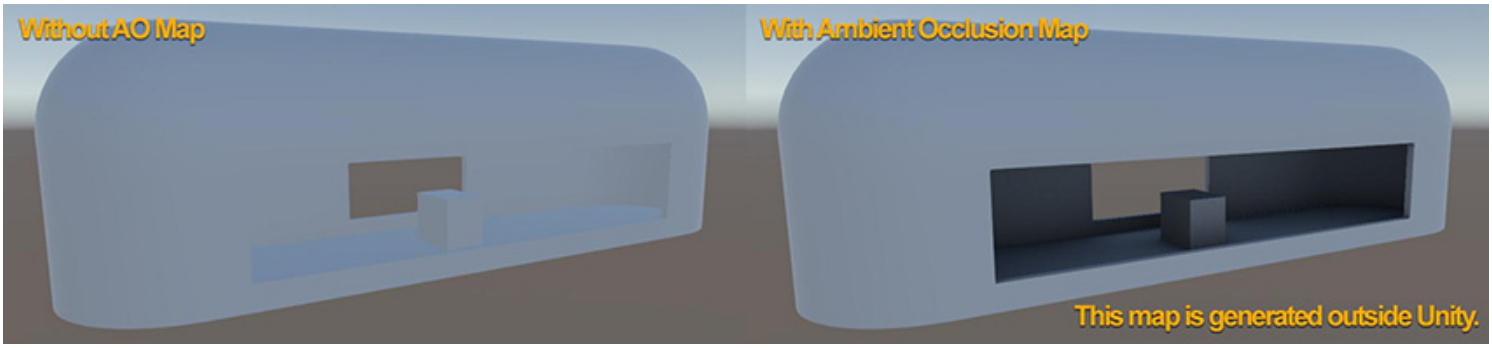
A normal map usually refers to a tangent space normal map that bends the surface normal of a polygon as if the light direction comes in from the other direction. This is usually used to add many details on a seemingly simple **mesh**.

You can also use normal maps to define a material. For example, you can show an original surface material for a piece of wood painted with high gloss red finish:

For more information on normal maps, see documentation on Normal maps (Bump mapping).

Occlusion map

An occlusion map mimics the reduction of ambient light, and can enhance the perception of concavity and shape.



Pre-baked Ambient Occlusion helps shade the ambient light

Using an occlusion map has the following benefits:

A detailed version of the occlusion map can be achieved at much higher quality during offline **rendering**, especially if the data is coming from a higher detailed model (similar to normal map baking from high-detailed model to low).

Occlusion maps help dynamically lit GameObjects tremendously, because dynamic GameObjects don't get occlusion from light baking, and only receive **Light Probe** or ambient lighting, and low detail Screen Space **Ambient Occlusion** (SSAO).

For more information on occlusion maps, see Occlusion Map.

Reference pictures, color charts and photo sources

Like trying to learn any new field, studying the general principles behind digital content creation improves your results. Taking pictures of a real surface, whether for a reference or for a texture source, often speeds up the creation of a surface material in a 3D modelling application.

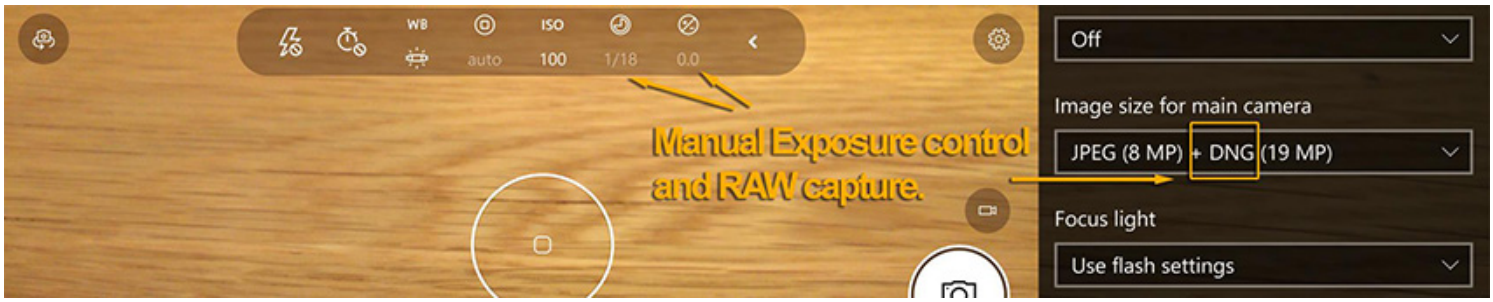
When capturing a reference, you simply need to take lots of pictures of the particular subject. It's the equivalent of going to an image search engine and searching for a specific reference image.

When taking pictures for a texture source, always neutralize the color cast and negate the light brightness effect of the surface. Use white balancing and proper exposure of an 18% gray card (such as Kodak R–27 card). For example:

The 18% grey card is a trusted value that you can use as an anchor when taking texture source pictures. Do not try to print your own 18% gray card, as this will most likely give you an inaccurate gray card and off balance color. A proper 18% gray card is equivalent to rgb(124,124,124) or #7C7C7C in sRGB color space. The **camera** uses this gray card to acquire the proper exposure (negating light brightness) and do white balancing (neutralizing the light color cast).

For more information on grey card, see Middle gray (Wikipedia).

For best results, use a Digital SLR / advanced camera. However, you can achieve a good result with the above setup on any camera, including a mobile phone with manual exposure control and RAW capability.



Example of a mobile phone setup to use manual exposure and RAW capture

An X-Rite ColorChecker-profiled RAW image with polarized lighting and lens produces higher accuracy in capturing an albedo texture.

Example of a ColorChecker chart

Like the R–27 gray card, the ColorChecker chart is a trusted color reference that you can use as an anchor. You generate a camera sensor + lens + filter + lighting condition characteristic profile, and use it to correct RAW images.

When capturing a texture source, choose diffused lighting conditions, such as a cloudy day, or evenly lit shades.

For purer albedo textures when the captured images are processed, you can apply additional processing using Unity's de-lighting tool.

2018–03–21 Page published with limited editorial review
Making believable visuals Best Practice Guide added in Unity 2017.3

Outdoor lighting and Scene setup

Leave feedback

At this stage, you should have meshes that are properly textured and an assembled Scene with proper tonemapped Unity render settings. But the Scene still needs a proper lighting setup. The advice in this section assumes you have set the Scene with a **Realtime GI** strategy and then lit with Realtime lights for instant feedback. The same advice and lighting principles also apply when baking.

For outdoor lighting and Scene setup, consider the following:

Environment lighting
Global Illumination
Light Probes and Reflection Probes
HDR Lighting Value
Screen Space Ambient Occlusion and Screen Space Reflection
Fog

Environment lighting

One of the main components in outdoor lighting is Environment lighting, which is also referred to as hemisphere lighting or skylight. Night sky has minimal contribution, while daytime sky has very bright contribution. Environment lighting settings can be found under the Lighting tab (**Window** > **Rendering** > **Lighting Settings**).

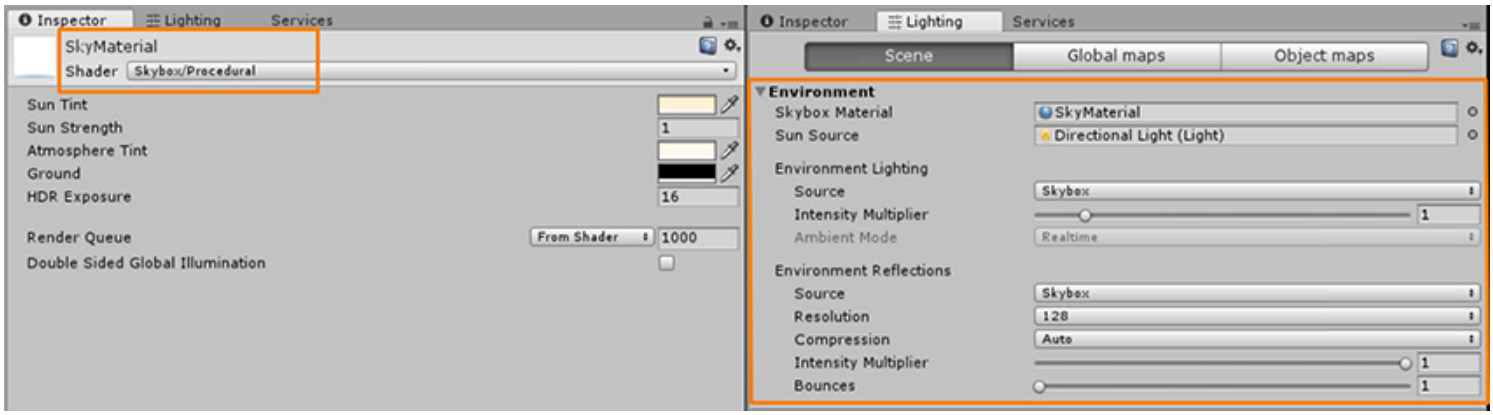
A procedural **Skybox** material is better than a **HDRI cubemap**, for its simplicity and ease of use.

To create a procedural Skybox material:

Add a new material in the project and name it **SkyMaterial**.

In the **Inspector**, set the material's **Shader** to Skybox/Procedural

In the Lighting window, on the **Scene** tab, assign **SkyMaterial** to **Environment** > **Skybox Material**:



The Scene is now slightly lit. It has ambient lighting, but not proper environment lighting:

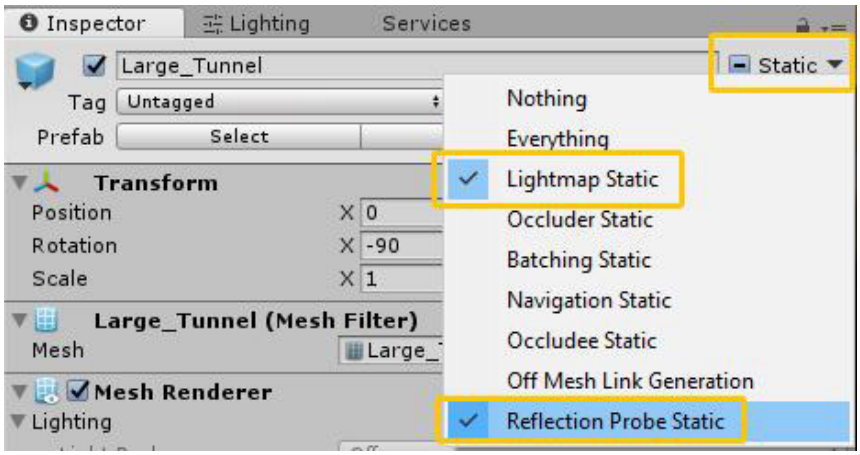
Directional Light

Typical sunlight or moonlight is usually represented by a directional light. This is due to the parallel nature of its light and shadow direction mimicking the light source at close to infinity distance.

Global Illumination (GI)

Directional and ambient light alone don’t create believable lighting. Proper environment lighting requires occlusion of the skylight lighting, and sunlight requires indirect lighting bounce. In our example, the sky currently renders a single color value to the Scene, making it flat. This is where you need Realtime GI or Baked lighting to calculate occlusion and indirect bounce lighting. To achieve that, follow these steps:

To make sure all the relevant meshes contribute to the Realtime GI or baking, select the **GameObjects** in your Scene and, in the Inspector window in the **Static** drop-down menu, enable **Lightmap Static** and **Reflection Probes**:



Note: You should generally apply this to all large static meshes.

Ensure that your Scene is set to use **Realtime Global Illumination** on the Lighting window > **Scene** > **Realtime Lighting** > **Realtime Global Illumination**.

At the bottom of the Lighting window, make sure **Auto Generate** is checked.

At this stage, the example Scene is now dark, and some elements of the Scene look out of place, such as the tram and the door on the background:

The static GameObjects in the Scene currently have proper occlusion for environment lighting (hemisphere lighting) and indirect bounce response from the directional light, but the rest of the GameObjects need a proper lighting setup.

Light Probes and Reflection Probes

For dynamic GameObjects and non-lightmap GameObjects to receive Realtime or Baked GI, you need to have Light Probes distributed in the Scene. Distribute **Light Probe groups** in the Scene efficiently, such as near dynamically-lit GameObjects or in areas where the player passes by. For more information on Light Probes group, see Light Probes.

Because **Auto Generate** is checked in the Lighting window, the lighting refreshes whenever you make a change. With Light Probes now in the example Scene, the tram and the background door are grounded better, but reflections look out of place. Sky reflection is all over the place and shows up inside the tunnel:

Reflection Probes help resolve this issue. Place Reflection Probes efficiently, with proper coverage in the Scene as needed. In the Scene image above, one Reflection Probe for the entire room is sufficient. A 128 **pixels** Cubemap Resolution using box projection is a good baseline for most cases, and is also useful for memory and reflection bake times.

The example Scene now looks properly grounded and cohesive, which is an important part of a believable Scene. However, everything is even darker than before, and nowhere near believable quality:

For more information on Reflection Probes, see Reflection Probes.

HDR Lighting Value

In reality, environment lighting and sunlights are very bright light sources. They're much brighter than pixel value 1. This is where HDR lighting comes into play, because it can accommodate pixel values outside the 0–1 range. To give you a good idea of what environment lighting does to a Scene, let's turn off the directional sunlight in our example Scene, and set the **SkyMaterial** > **Exposure** to 16:

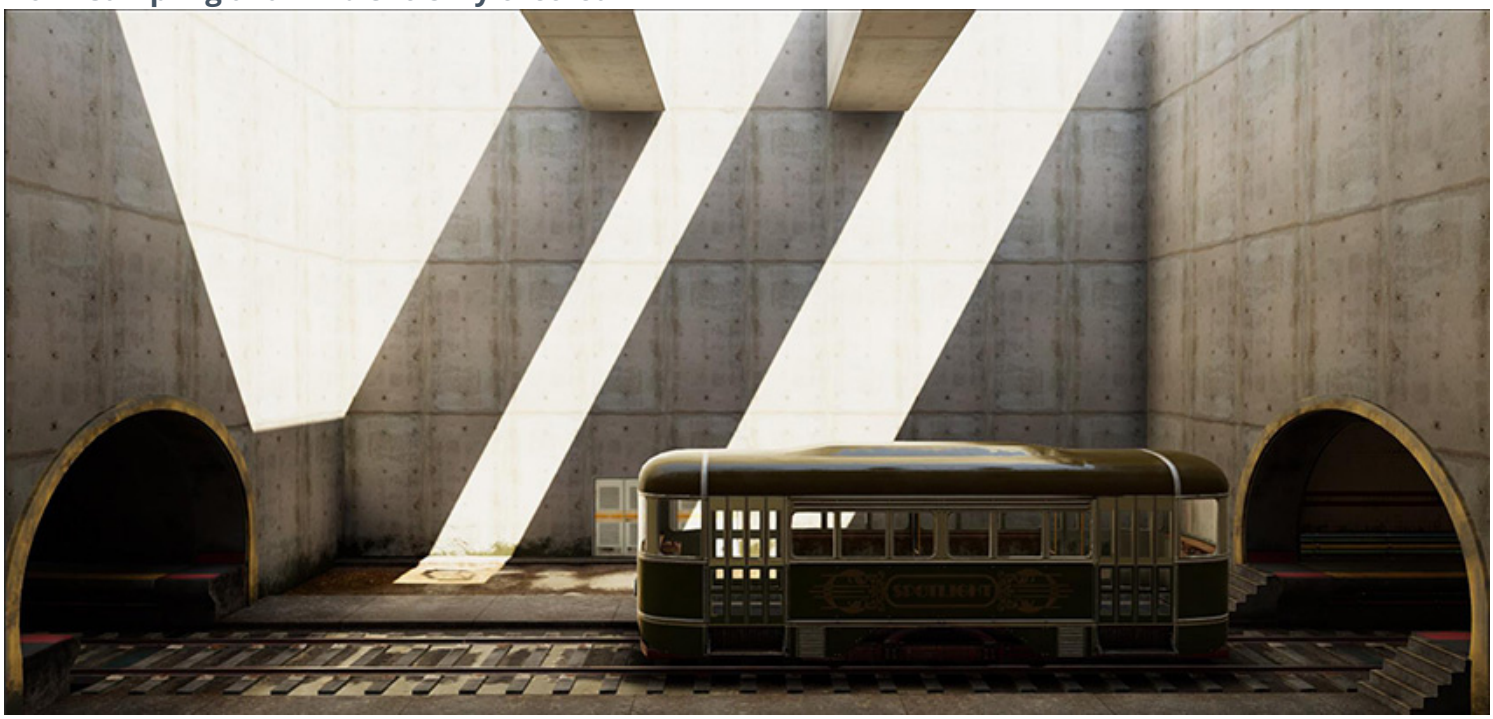
The Scene starts to look believable, as if it is a cloudy day when sunlight is completely diffused in the sky and directional light doesn't show up. Now, let's reintroduce sunlight back into the Scene at a much higher value by setting the **Intensity** to 5. Despite the sun looking nearly white, it is important that you choose color properly, because the impact of indirect color from a strong sun can dramatically change the look of the Scene.

Now the sun (directional light) looks like a high energy light, as expected from real life, and the Scene looks quite believable:

Screen Space Ambient Occlusion and Screen Space Reflection

While the Scene lighting looks pretty good at this point, you can add more details to make the Scene more believable. Baking detailed occlusion usually isn't possible because of the limited resolution set in the Realtime GI for reasonable performance. This is where Screen Space Ambient Occlusion can help. Enable SSAO in the Post-Process Profile by checking **Ambient Occlusion**. The sample Scene contains a post-processing profile called *Rin_Post_Basic.asset*, set to the following **Ambient Occlusion** settings:

Intensity 0.5
Radius 1
Sample Count Medium
Downsampling and **Ambient Only** checked

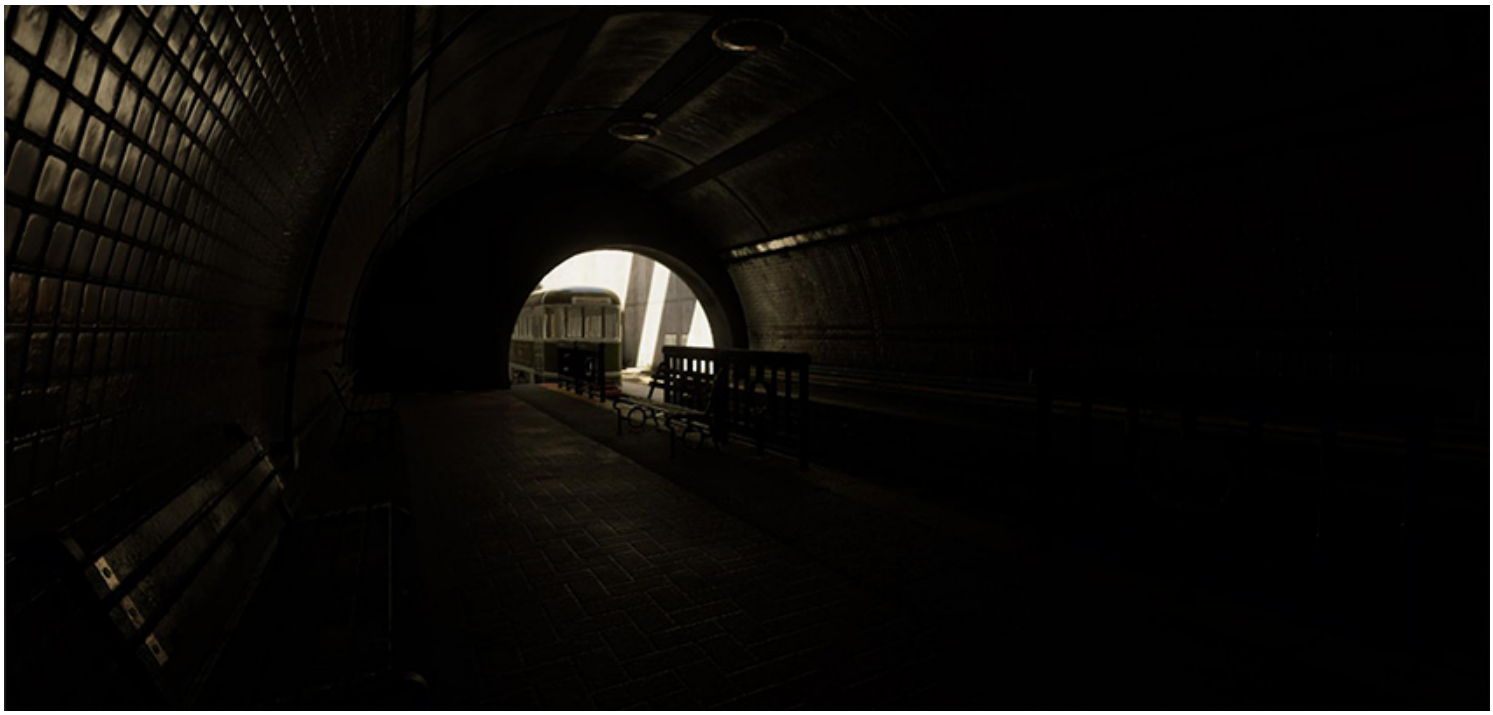


While SSAO takes care of extra ambient lighting occlusion, Screen Space Reflection improves the accuracy. When you enable the **Screen Space Reflection** in the post-processing profile, the left side of the wet track no longer renders bright reflections, because SSR gives the Scene more accurate reflections for on-screen GameObjects:

Both of these post-processing options incur performance impacts at run time, so enable them wisely. Adjust the values so that they have a reasonable performance impact that fits your run-time requirements.

Fog

At this stage, our sample Scene has believable outdoor and indoor value separation on a fixed exposure. Reflection is visible in the dark indoor areas as strong highlights and not dim muddy values. However, the Scene foreground and background elements are not showing up, despite being strong perspective elements:



Scene showing no differentiation between foreground and background scene elements

Adding a subtle fog to the Scene creates a massive difference in giving the Scene another dimension:

Scene showing exaggerated fog that creates separation between foreground and background elements

With fog added, the foreground railing is defined better compared to the zero fog Scene above. To enable fog, open the Lighting window and go to **Scene** > **Other Settings > Fog**. In the provided Scene, **Fog** is set to use **Color** #6D6B4EFF, and it's in **Exponential** mode with **Density** set to 0.025. In deferred rendering, you might need to enable fog in the post-processing profile if it is not activated automatically.

2018–08–21 Page published with limited editorial review

Making believable visuals Best Practice Guide added in Unity 2017.3

Indoor and local lighting

Leave feedback

For indoor and local lighting, consider the following:

Spot lights and Point lights
Color and Intensity of light
Emissive surfaces

Spot lights and Point lights

The staples of real-time local lighting are spot lights and point lights. Fundamentally, both of these types of lights emit light from one point in space and have a limited range. The spot light has an additional limit by angle. For more information, see documentation on Types of light.

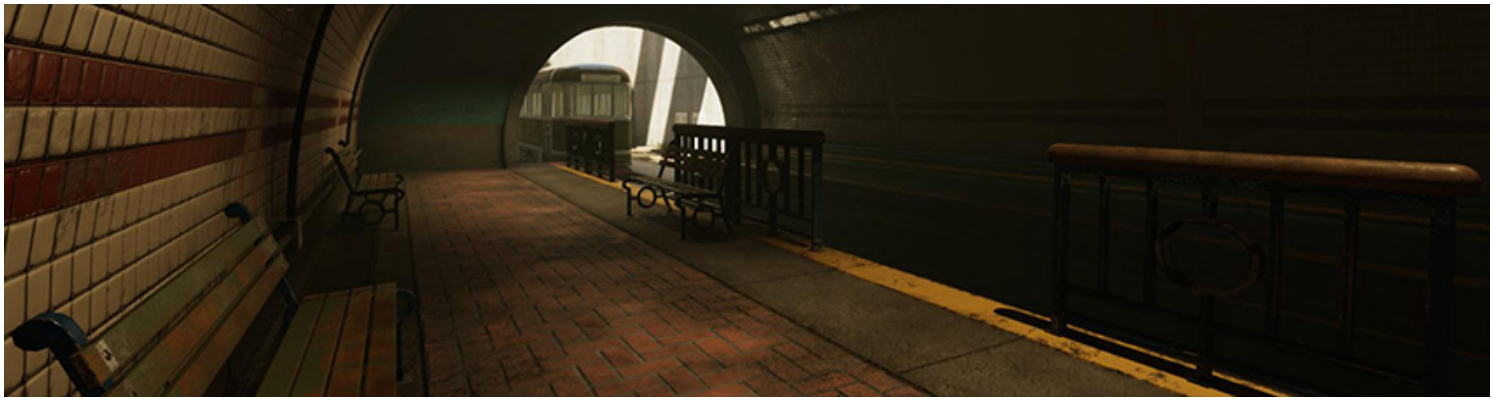
The main difference between spot lights and point lights is how they cast shadows and interact with cookies. Shadowing with a point light requires 6 shadow maps, while a spot light only needs one. For this reason, point lights are much more performance-intensive, and you should use them sparingly. Note that **baked lights** do not require shadow maps.

Another difference is that a cookie texture on a spot light is a simple 2D texture, while a point light requires a cube map, usually authored in a 3D modelling application. For more information, see documentation on Cookies.

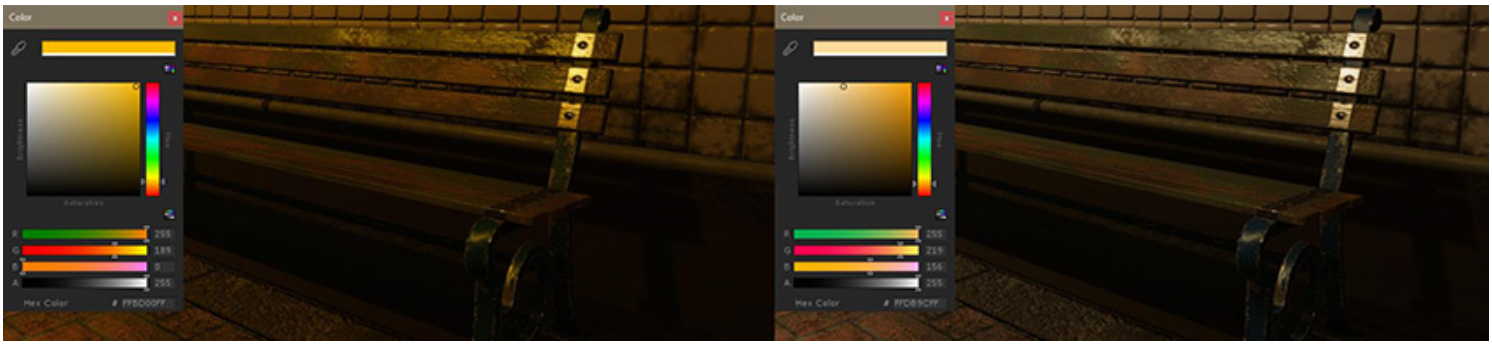
Note: You can only use area lighting when baking lighting, unless you're using the HD Scriptable Render Pipeline. There are area lights that can be rendered in real time when in HD SRP mode.

Light intensity and color

When selecting intensity for indoor lights, try to make sure no indoor lights have a greater intensity than the sun's light. The sample **Scene** is set in a tunnel, so it's very unlikely that there are any high intensity lights shining from the ceiling that exceed the brightness of the sun's light:



When choosing the right color for your lights, consider the effect of the color and value chosen. When selecting color, try not to completely leave out any one of the color channels completely. This creates a light that is difficult to converge with the white point. For example, although it is technically a valid light color, the light color on the left image below removes all blue color from the final output:

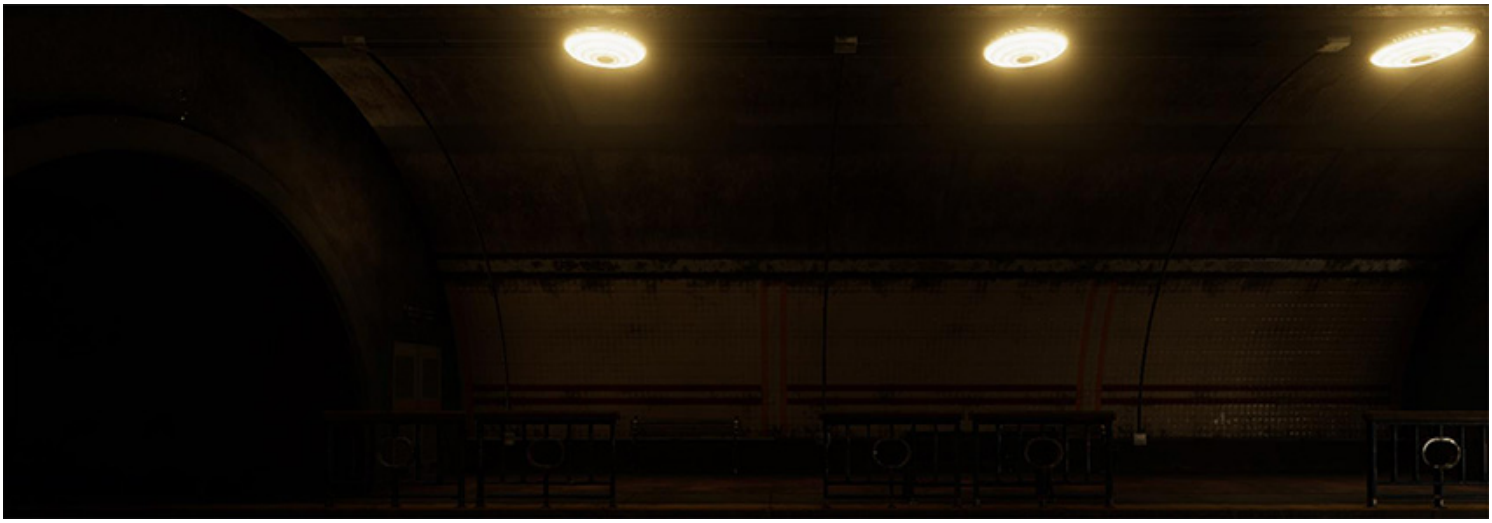


Try not to limit your final color palette in the Scene, especially if you expect to do color grading later on.

Note: For advice on using a tonemapper to handle high intensity color values, such as colored light or fire, see High intensity color.

Emissive materials

In Unity, emissive materials can contribute to lighting if **Realtime GI** or **Baked** lighting is enabled, giving the effect of area lighting. This is especially useful if **Realtime GI** is enabled. You can modify the intensity and color of the emissive surface and get feedback immediately, assuming that pre-computations have been done ahead of time.



Small meshes on the ceiling that use the Standard Shader in **Emission** mode

Subtle diffuse lighting coming from small meshes on the ceiling

For more information, see Emission.

At this point, you should have a good understanding of how to set up and light a Scene to look believable.

2018–03–21 Page published with limited editorial review

Making believable visuals Best Practice Guide added in Unity 2017.3

Understanding post-processing

Leave feedback

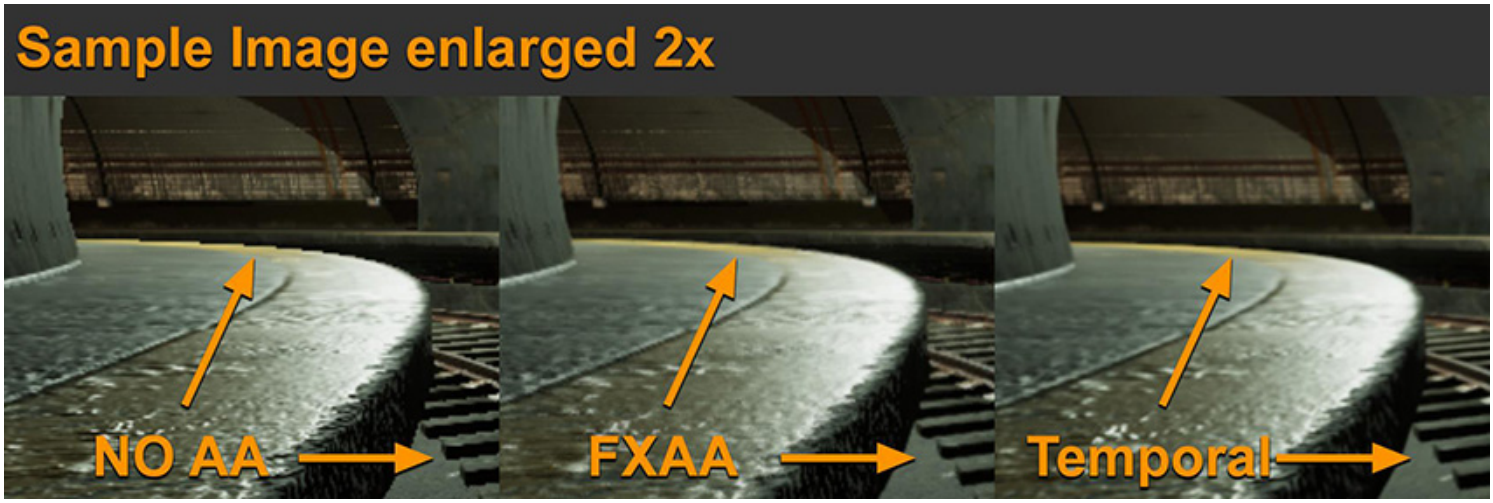
Post-processes are **rendering** effects that are based on an existing rendered **Scene**. Effects in post-processing are usually dependent on the **Scene view**, or layered on top of the rendered **Scene** before generating the final render. The clear advantage of this feature is the instant visual feedback and dramatic improvement to the **Scene**, without the need to alter existing content.

You don't need to use all of the post-processing features to create believable Scenes. However, its capability to enhance a Scene further is certainly worth the time it takes to understand the system. This section helps you decide which post-processing effects are right for your situation, and helps you avoid the pitfalls that can come with these advanced features:

Anti-aliasing
Ambient occlusion (AO)n
Screen Space Reflection
Depth of Field
Motion blur
High intensity color
Chromatic Aberration, Grain and Vignette
For an in-depth overview on post-processing, see the Post-processing documentation.

Anti-aliasing

When rasterizing a 3D polygon into a 2D screen with limited resolution, the final **pixels** show an aliasing effect (like steps on the edges of objects). There are different solutions for anti-aliasing techniques in real-time 3D, such as supersampling, multisample anti-aliasing (MSAA), fast approximate anti-aliasing (FXAA), spatial anti-aliasing (SMAA) and temporal anti-aliasing (TXAA). Currently, the most popular techniques are FXAA and TXAA, due to their effectiveness and relatively high performance. For example, the following image from the sample Scene compares FXAA and TXAA:



FXAA does a good job of fixing some of the glaring aliasing effects, but TXAA takes it a step further and performs a much better job in the tram rails.

FXAA is pure post-processing anti-aliasing. In simple terms, It captures the rasterized Scene, analyzes the edges, and runs an algorithm on top of the existing image to smooth it out. It is straightforward, doesn't have any complex dependencies, and it's fast.

TXAA is a lot more complex. It uses jittering and the previous frame as additional data to blend into the current frame. It also uses motion vectors to predict which pixels to reject or accept to render the final frame. The idea is to increase the effective resolution of a frame with more data, without the need to render the Scene larger than its final resolution (as is usually the case in supersampling). The benefit is clearly a much smoother anti-aliasing, similar to the quality given by supersampling, but without the major performance impact.

Like everything in real-time rendering, there's always a trade-off. TXAA requires motion vectors to function, and has a larger performance impact when compared to FXAA. TXAA's complex nature of predicting the final image can cause some unintended artifacts for fast-moving **GameObjects**, and texture blurriness across the Scene, which might not be suitable for some applications.

Ambient occlusion (AO)

Ambient occlusion post-processing is an approximation of **ambient occlusion** based on screen space data, mainly depth. For this reason, it is usually called Screen Space **Ambient Occlusion** (SSAO). As explained in Enable post-processing, SSAO can give better fidelity when shading ambient lighting, especially for dynamic GameObjects that often don't have any occlusion interaction between static Scenes and dynamic Scenes.

Screen Space Ambient Occlusion - Debug View

While in general SSAO helps a Scene's ambient shading, it can cause too much occlusion. Using per-object baked ambient occlusion from an offline 3D modelling application with additional ambient occlusion from light baking makes SSAO a third layer of ambient occlusion.

Make sure you keep the final output in mind when setting up SSAO, and try to find a balance with the other ambient occlusion solutions.

Adding too much ambient occlusion can cause open areas to be very dark

Screen Space Reflection

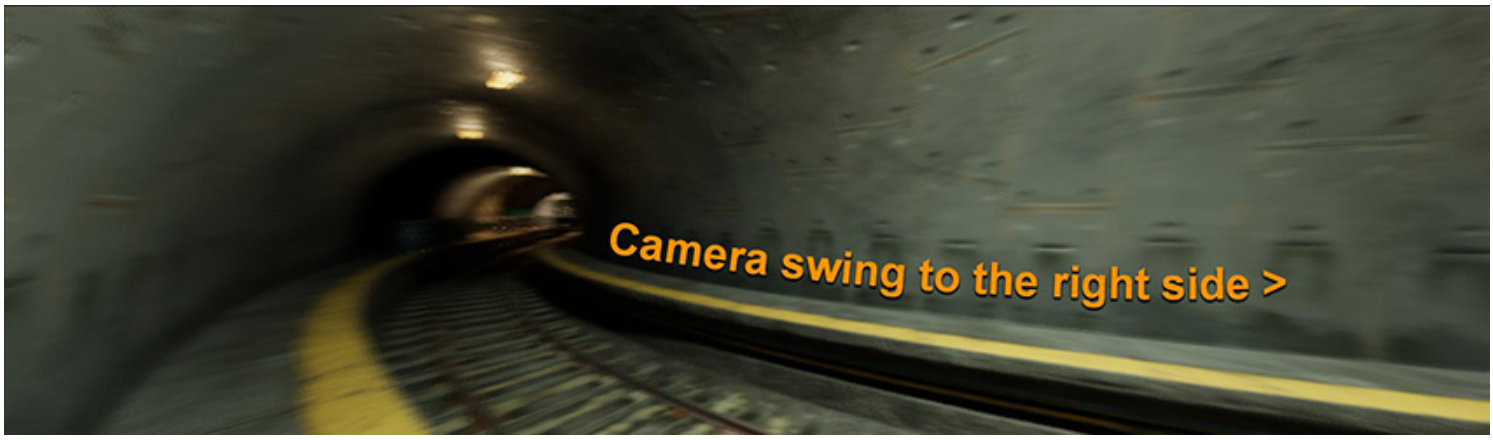
Like SSAO, Screen Space Reflection uses the current Scene view to approximate reflections via ray tracing. To get believable results, it is almost always a good idea to enable this feature. It adds a highly accurate reflection that complements the normal cube map captured reflection. However, enabling this feature does restrict rendering to deferred rendering only, and has a performance impact. Another downside of SSR is that anything not on the screen does not generate reflection hits, and therefore can cause a missing reflection effect at the edges of the screen.

Depth of Field

The Depth of Field effect can give the cinematic feel of a large sensor **camera**. It can also be used to change the scale perception of a Scene (much like how a tilt-shift camera lens gives a miniature effect). For example, the following image is a real-life photograph, made to look like a miniature by applying a Depth of Field effect:

Motion blur

A subtle motion blur can make blending the transition of one frame to another more believable. However, you need to take care in your application to reduce the risk of motion sickness. This is especially true if there is a massive difference in Scene translation, which is typically found in first-person or third-person cameras. For example, a wide-angle view where the player can swing their camera really fast from left to right looks jittery and give a stop motion look without motion blur, even if it is rendering at 60FPS. The following image shows a running motion blur at a Shutter angle of 180 degrees:

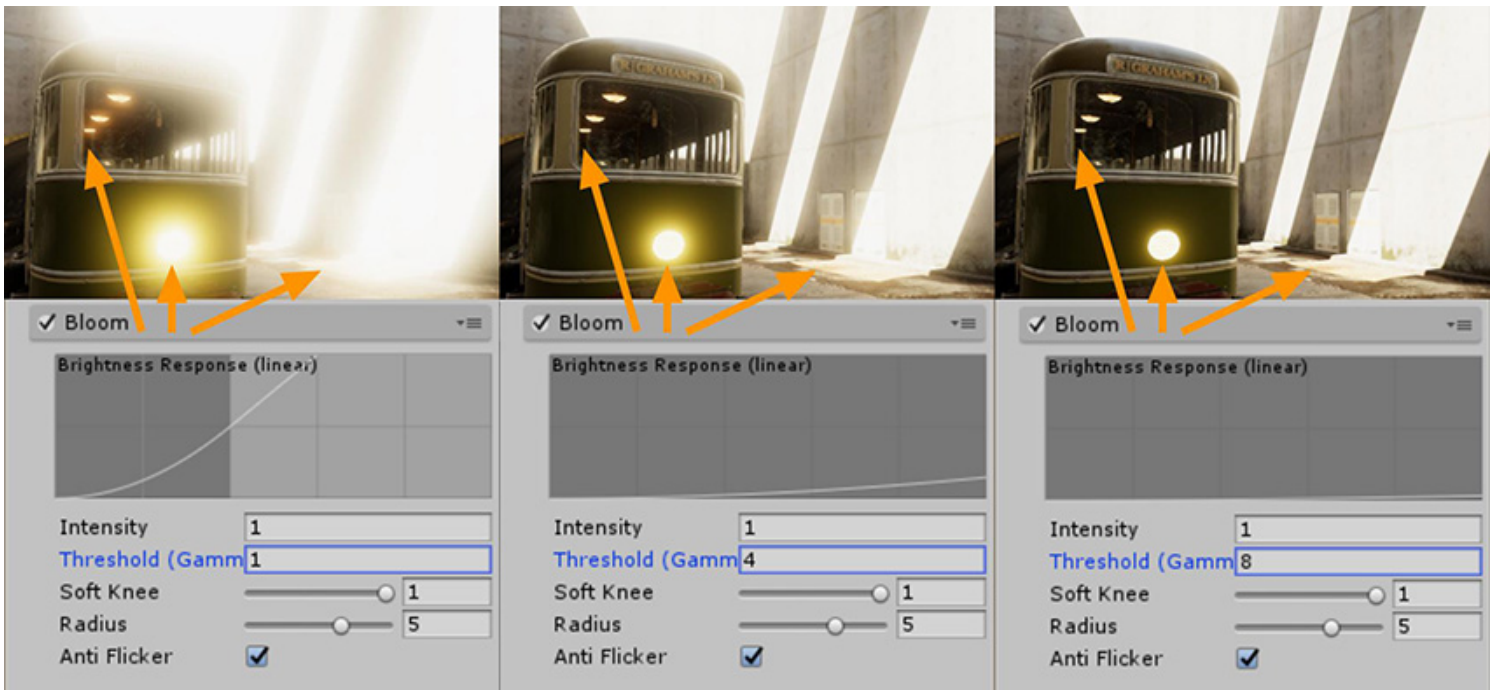


A full 360-degree shutter angle gives you a full frame duration trail. Anything less means less trail. With that in mind, if you are aiming for a stop-motion look, then disable motion blur.

Bloom and emissive

In real life, bloom is a lens artifact where light beams aren't focused properly. It usually occurs on lower-quality camera lenses, or some special effects glow camera filter. Bloom gives a hazy soft image, as shown here (**Threshold Gamma** set to 0):

Or you can use Bloom to differentiate elements of high intensity or bright light, as shown here:



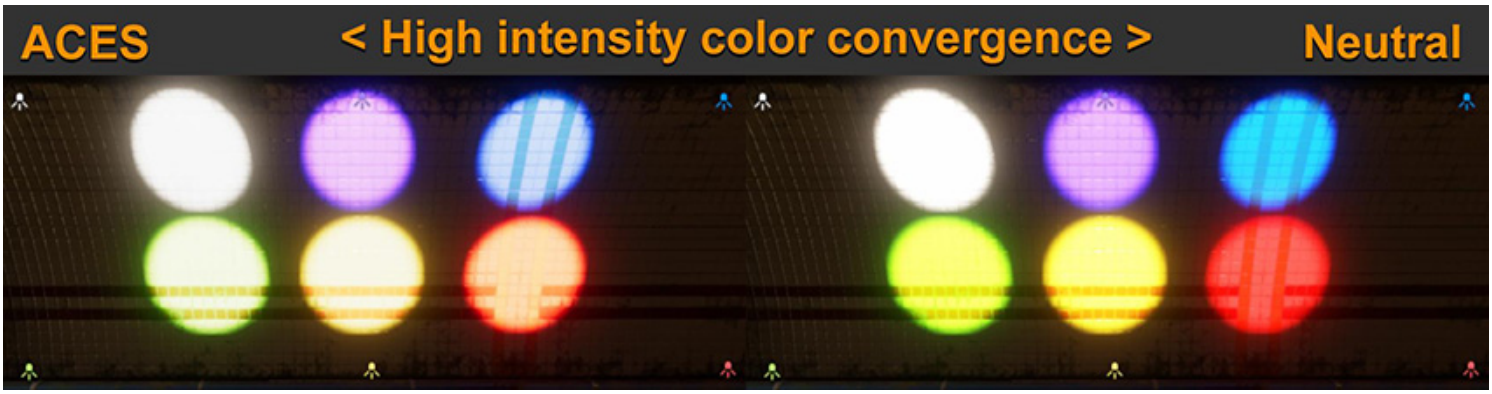
Over-using these features can backfire. You can see this happen in the first sample in the above image; there are lots of high intensity pixels, and the threshold of the intensity starts to bloom very early.

When you select the value of the threshold, consider the specific values of your emissive surfaces, the lighting setup for the Scene, and whether you have enabled eye adaptation.

High intensity color

You can handle high intensity color values using a tonemapper, which processes a linear **HDR** buffer of input data and renders it back out to the designated color space for final output. This is similar to how a camera works. In Unity post-processing, there are two types of tonemapper: **Neutral** and **ACES** (Academy Color Encoding System (Wikipedia)). At first glance, the difference between the two is in the default contrast of the tonemapper. However, the main difference between the two is that you can adjust **Neutral** to be similar in contrast to **ACES**. In the image below, the two examples are almost identical (where the **Neutral** settings are: **Black** In 0.02, **White** In 10, **Black Out** –0.04, **White Out** 10, **White Level** 5.3 and **White Clip** 10):

The main difference you need to take into account is how the two tonemappers handle high intensity color values, such as colored light or emissive effects from explosion effects or fire. The following image shows how an **ACES** tonemapper normalizes high intensity color differently to the **Neutral** tonemapper:



Chromatic Aberration, Grain and Vignette

Chromatic Aberration, Grain and Vignette are post-process effects that simulate artifacts from real-life camera systems. To use them properly, it is good to understand how each of these occur in real cameras:

Chromatic Aberration (CA) is a dispersion of color that appears on an image if the lens of a camera fails to focus all color to the same convergent point. This is usually found in a poorly calibrated or lower quality lens. While this can sometimes add a sense of realism to a digital Scene, this also means your virtual camera is directed to convey a low quality lens.

Grain seen in the final image of a real photograph or cinema is usually a sign of an insufficient quantity of useful light entering the sensor, such as a dark scene or a high ISO camera sensor/film translating to noise. You can use this effect to simulate camera limitations, to make a pristine 3D-rendered Scene feel more believable. However, too much noise in a Scene can distract the viewer with a false sense of motion, and affect the contrast of the final rendered image.

Vignette is similar to the CA effect; it adds artifacts where a lens could not give consistent light coverage from the center to the edge of the sensor/film of a camera. You can use this effect to give some sense of focus for a central point of a Scene. However, if you use this effect too much, you can make a Scene look like it was processed by an amateur post-editor.

The key takeaway from these post-processing fundamentals is that you should practice and develop an understanding of each effect, so that you can effectively use the features with a sense of purpose, and weigh the visual benefit of each additional effect against overall Scene performance.

For advice on additional post-processing features, see documentation on:

Eye Adaptation

Color Grading (other than tonemapper)
User Lut (Lookup Table)

2018–03–21 Page published with limited editorial review
Making believable visuals Best Practice Guide added in Unity 2017.3

Dynamic lighting

Leave feedback

Dynamically-lit **GameObjects**, especially large **GameObjects**, require more tricks than their static counterparts. **GameObjects** usually require dynamic lighting if they change position in the **Scene**, so pre-calculated lighting isn't an option. However, dynamic **GameObjects** have to work within certain limitations. To improve the quality of dynamic GameObject lighting, consider:

Light Probe Proxy Volume (LPPV)
Per-object baked Ambient Occlusion Map (AO)
Local Reflection
Fake Shadows or occlusion based on assumptions

Light Probe Proxy Volume (LPPV)

The surfaces of dynamic GameObjects that aren't lit by dynamic lighting generally use **Light Probe** data to fill in their lighting information (in a Scene where Light Probes are not present, Environment Lighting is used). Depending on the lighting strategy used in a Scene, this can range from indirect lighting information to shadowing and baked diffused Light Probe lighting information. This Light Probe strategy usually works fine for small dynamic GameObjects. However, larger GameObjects require a finer granularity of Light Probe lighting. This is where **Light Probe Proxy Volumes** come in. For more information, see Light Probe Proxy Volume component.

Using Light Probe Proxy Volumes allows a large dynamically-lit GameObject to use more than a single Light Probe, resulting in higher lighting accuracy. The following example shows how the capsule with LPPV demonstrates higher accuracy of Light Probe sampling, despite only using 2x2x2 Volume grid:

Capsule comparison showing impact of LLPV

Per-object baked Ambient Occlusion Map (AO)

Dynamic GameObjects only receive lighting from Light Probes or ambient lights. So you need to pre-calculate an occlusion for the GameObject, especially if the GameObject involves a concave interior, such as the tram in the example. In the example below, the tram on the left without AO is applying Light Probe lighting data without knowing how to differentiate the interior and the exterior surfaces. With the pre-baked AO, this map serves as a guide to reduce the intensity of light and reflection from the exterior, giving a much more grounded look:

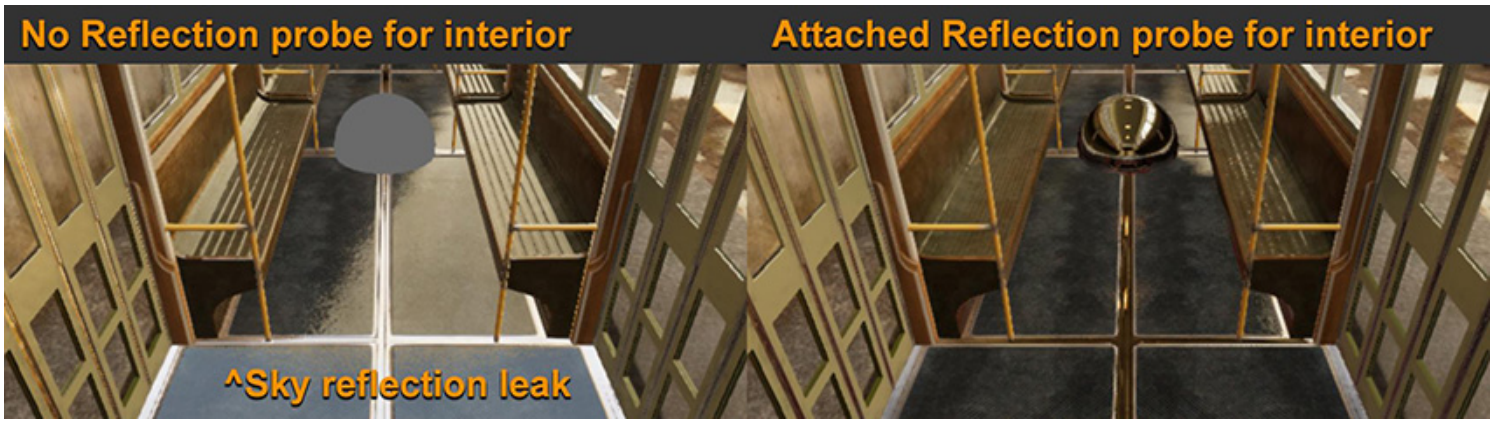
Tram comparison showing impact of AO

Per-GameObject AO offline baking can even give further detailed occlusion by baking from a higher detailed **mesh** to lower a detailed **mesh**, in a similar way to how normal map baking works.

Note: Per-GameObject AO doesn't interact with other dynamic GameObjects. For example, a dynamic GameObject (such as character entering the tram) receives Light Probe data from the Scene and doesn't necessarily match the occlusion of the tram interior.

Local reflection

Most dynamic GameObjects don't need their own reflection. However, for GameObjects that involve concave interiors, attaching a **Reflection Probe** to the GameObject and allowing it to run in real time can help reduce false reflection hits coming from the environment **Reflection Probe**.



Exaggerated material showing reflection issues

Fake shadows or occlusion based on assumptions

If you can make certain assumptions for a GameObject, there are tricks that you can use to improve visual quality. In the sample shown below, the tram should always be on the rails. So to help its ground light occlusion in shaded areas, we can use a "Particle/Multiply" transparent material plane:

Simple Plane trick using a Particle/Multiply transparent material

A similar trick is to place a blob shadow projector under a character instead of the character casting real shadows.

In real-time **rendering**, if you can find a trick that works, and it's cheap on performance, it's usually a viable solution. There are certainly more tips and tricks that improve visual **rendering**. The above list should give you confidence in thinking of solutions for different kinds of visual requirements.

2018–03–21 Page published with limited editorial review
Making believable visuals Best Practice Guide added in Unity 2017.3

Best Practice: Setting up the Lighting Pipeline

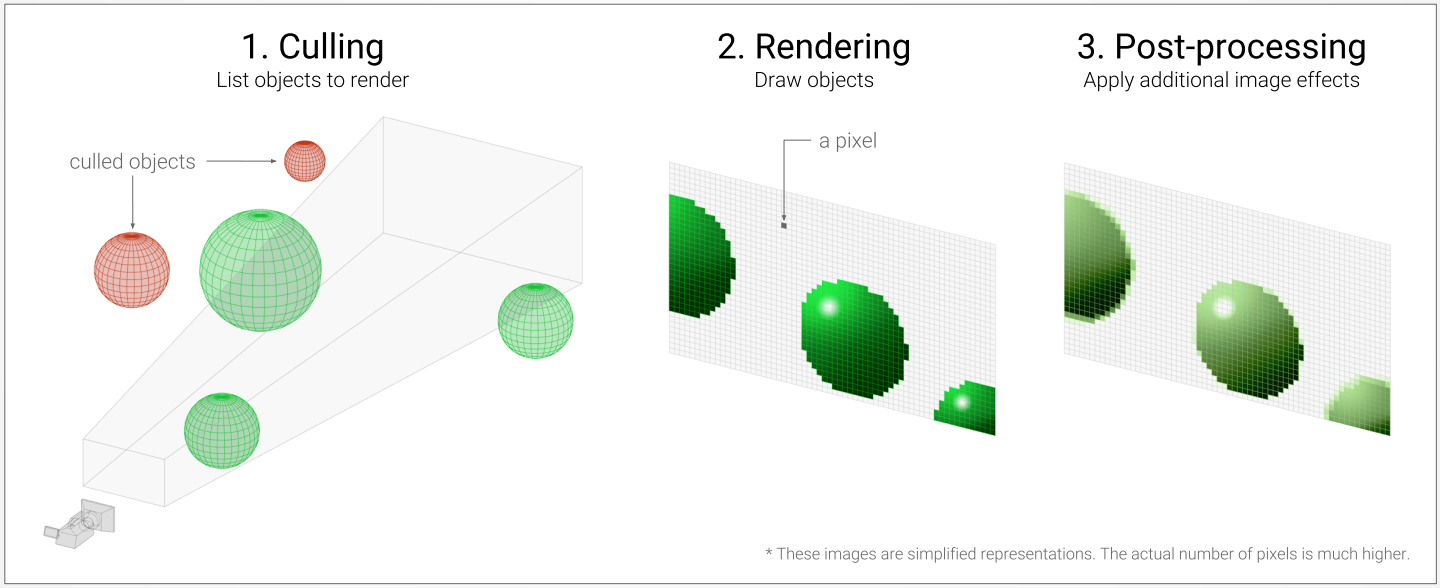
Leave feedback

Definitions

First, let's go through the definitions of several important graphics **rendering** terms that you will encounter frequently in this guide.

A **render pipeline** determines how the objects in your **scene** are displayed, in three main stages.
The first step is **culling**; it lists the objects that need to be rendered, preferably the ones visible to the **camera** (frustum culling) and unoccluded by other objects (occlusion culling).
The second stage, **rendering**, is the drawing of these objects, with the correct lighting and some of their properties, into pixel-based buffers.
Finally, **post-processing** operations can be carried out on these buffers, for instance applying color grading, bloom and **depth of field**, to generate the final output frame that is sent to a display device. These operations are repeated many times a second, depending on the frame rate.

Spotlight On
Render pipeline stages



A **Shader** is a generic name for a program, or a collection of programs, running on the graphics processing unit (GPU). For instance, after the culling stage is completed, a **vertex shader** is used to transform the vertex coordinates of the visible objects from "object space" into a different space called "clip space"; these new coordinates are then used by the GPU to rasterize the scene, i.e. convert the vectorial representation of the scene into actual **pixels**. At a later stage, these **pixels** will be colored by pixel (or fragment) shaders; the pixel color will generally depend on the material properties of the respective surface and the surrounding lighting. Another common type of shader available on modern hardware is **compute shaders**: they allow programmers to exploit the considerable parallel processing power of GPUs for any kind of mathematical operations, such as light culling, particle physics, or volumetric simulation.

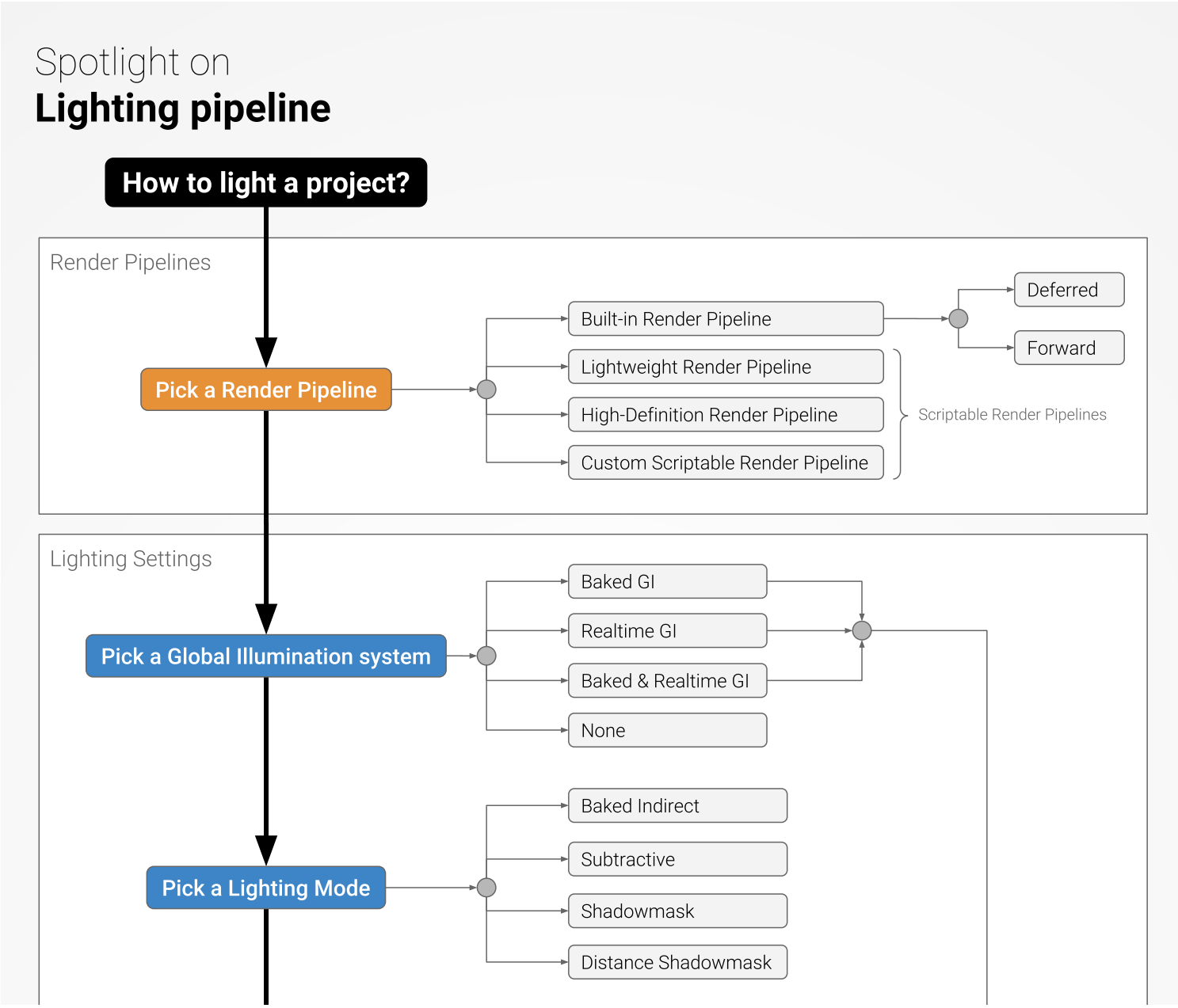
Direct lighting refers to lighting which originates from a self-emitting source of light, such as a light bulb, and isn't the result of light bouncing off a surface. Depending on the size of the light source and its distance to the receiver, such lighting typically produces clear distinct shadows.
Direct lighting should not be confused with **directional lighting**, which is light emitted by an infinitely-distant light source (e.g. the computer-simulated sun). The noticeable properties of a directional light are the ability to cover the entire scene with parallel light rays, and the lack of distance falloff (or light decay); that is, the amount of lighting received does not decay as the distance to the light source increases.
In reality, the sunlight, like any other source of light, falls off over distance, based on the inverse-square law. Simply put, the amount of received light drops very quickly when increasing the distance between the receiver and the light source. For example, the illuminance on Mercury is almost 7 times higher than on Earth, and Mars receives nearly half of Earth's sunshine, whereas Pluto enjoy a mere 0.06%. Nevertheless, for most real-time applications with a very

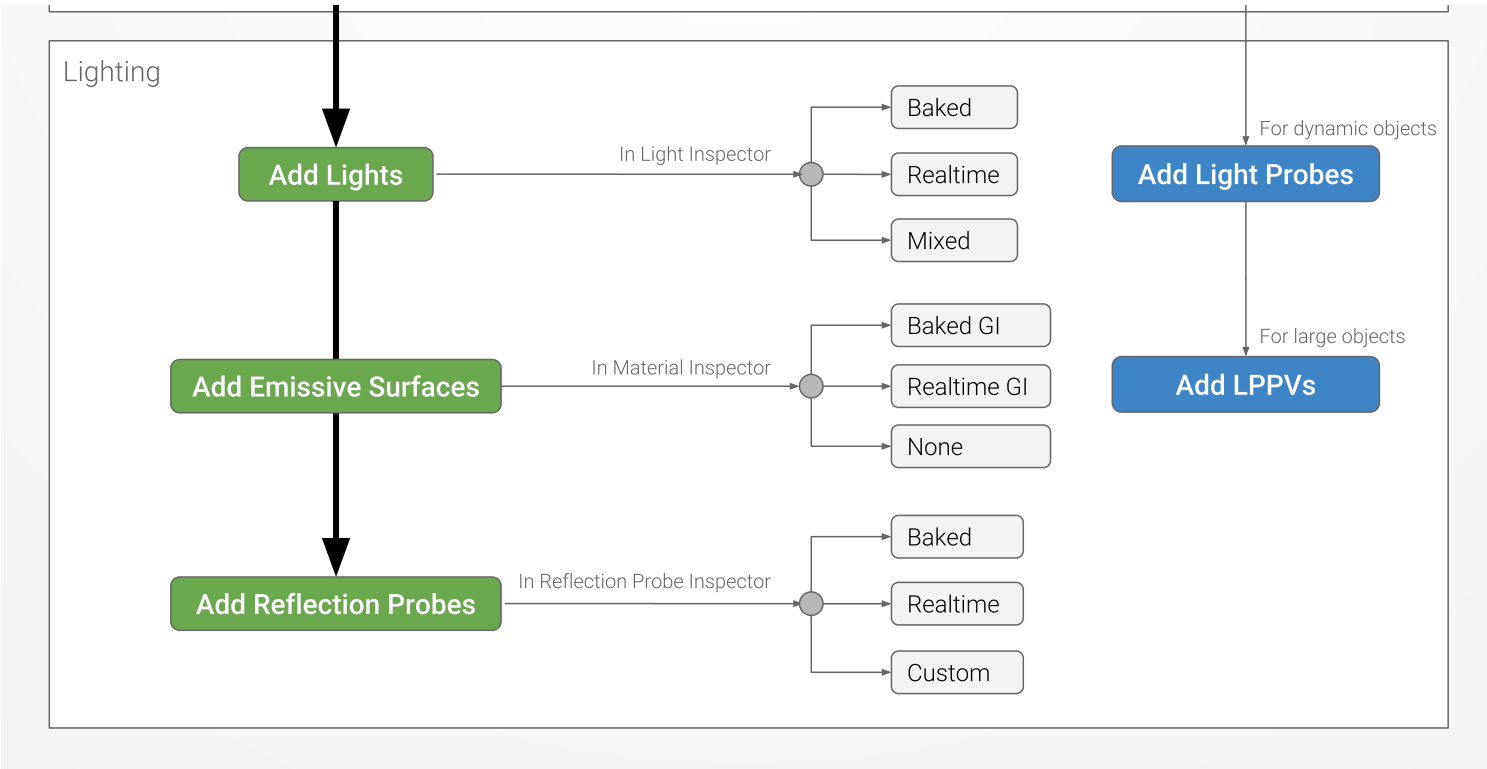
limited altitude range, the sunlight decay is insignificant. Therefore, the directional light is perfectly adequate to simulate sunlight in most Unity scenes, including large, planet-centric, open worlds. For other types of light such as Point and Spot, Unity now offers physically-based falloffs when you choose the High-Definition Render Pipeline.
Indirect lighting results from light bouncing off surfaces and being transmitted and scattered through a medium, such as the atmosphere or a translucent material. Under these conditions, occluders generally cast soft to indiscernible shadows.

Global illumination (GI) is used to generate the indirect lighting in the scene, primarily as a function of the direct lighting. There are several methods for GI, such as baked/dynamic **lightmaps**, irradiance volumes, light propagation volumes, baked/dynamic light probes, voxel-based GI, and distance field-based GI. Out of the box, Unity supports baked/dynamic **lightmaps** and light probes.
A **lightmapper** is the underlying system that generates the data for the lightmaps and light probes by shooting light rays, calculating the light bounces, and applying the resulting lighting into textures. Different lightmappers will therefore often produce different lighting looks, as they might rely on different techniques to produce the lighting data. Currently, Unity offers two lightmappers: Enlighten and the Progressive **Lightmapper**. Many more details about the GI systems and their lightmappers will be provided in an upcoming section of this guide.

Overview

The following flowchart provides a high-level perspective of the entire lighting pipeline in Unity, from the point of view of a content creator.





You start by selecting a render pipeline. Then you decide how the indirect lighting is generated and pick a Global Illumination system accordingly. After you've made sure all the global lighting settings are tuned appropriately for your project, you can continue adding Lights, Emissive Surfaces, Reflection Probes, Light Probes, and Light Probe Proxy Volumes (LPPVs). Detailing the usage and features of all these lighting objects is beyond the scope of this article, therefore I encourage you to read the Lighting section of the manual to learn how to utilize them correctly in your projects.

Render pipelines

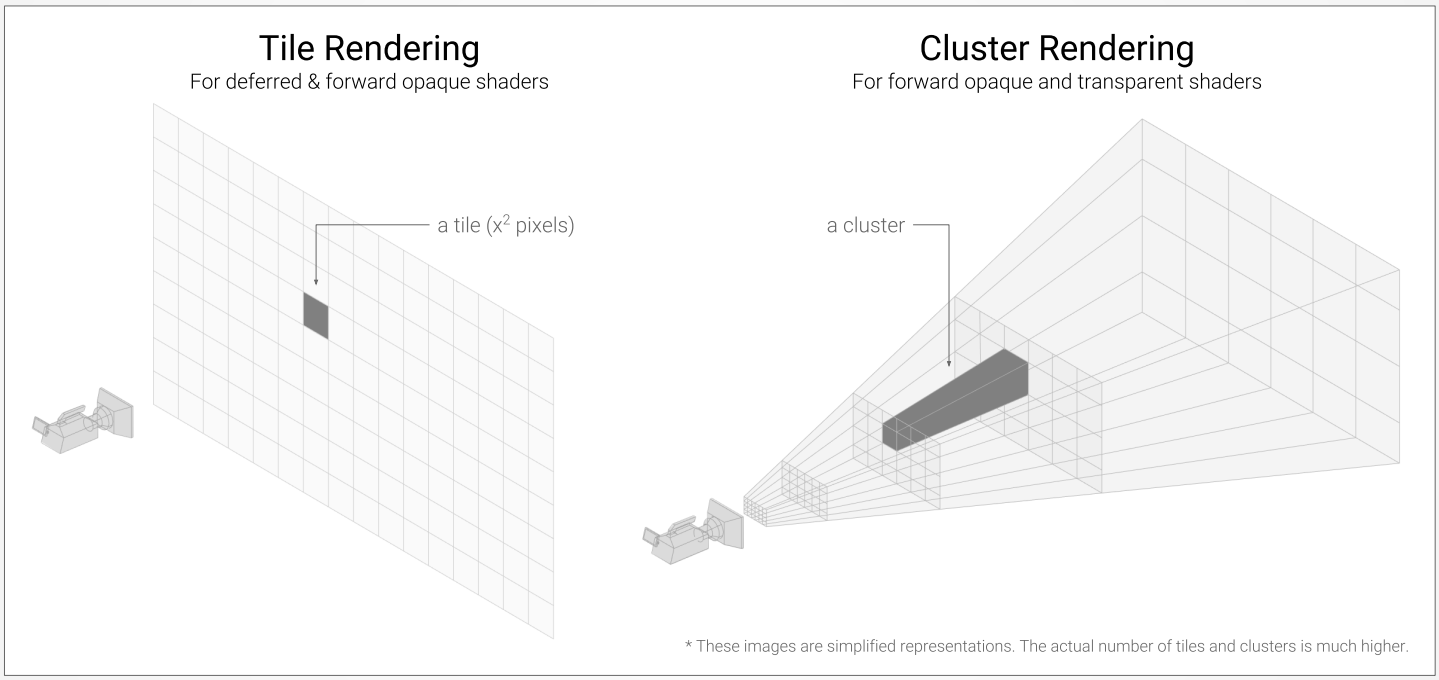
Until early 2018, only one render pipeline was available in Unity; it has been renamed the "Built-In Render Pipeline." This renderer offers a choice between forward and deferred rendering.

In (multi-pass) forward mode, all objects in the scene are rendered one by one sequentially, potentially in multiple passes, depending on the number of lights affecting each object, thus the rendering cost can dramatically increase when objects are lit by multiple lights. This type of renderer commonly offers a wide variety of shaders and can handle transparency easily.
In deferred mode, all the (opaque) geometries are first rendered into buffers that store information about their materials (color, specular, smoothness, etc.). In a later pass (hence "deferred"), each pixel is shaded sequentially: the rendering time will depend mainly on the number of lights affecting each pixel. The transparent objects, and certain objects with complex shaders, will still require additional **forward rendering** passes. Deferred rendering is usually recommended when dealing with scenes containing many dynamic lights, such as artificially lit interiors or projects with a combination of outdoor and indoor lighting.
In January 2018, we unveiled the Scriptable Render Pipeline (SRP), which allows you to customize the rendering loop via C# scripting. This is actually a minor revolution in the realm of game engines: users are finally able to personalize the culling of objects, their drawing, and the post-processing of the frame without having to use a low-level programming language like C++.

Unity currently provides two preview SRPs that are designed with performance in mind and for modern hardware:

The High-Definition Render Pipeline (HDRP) is a hybrid deferred/forward tile/cluster renderer. It offers advanced rendering and shading features and is designed for PC and console projects that require a high degree of visual fidelity.

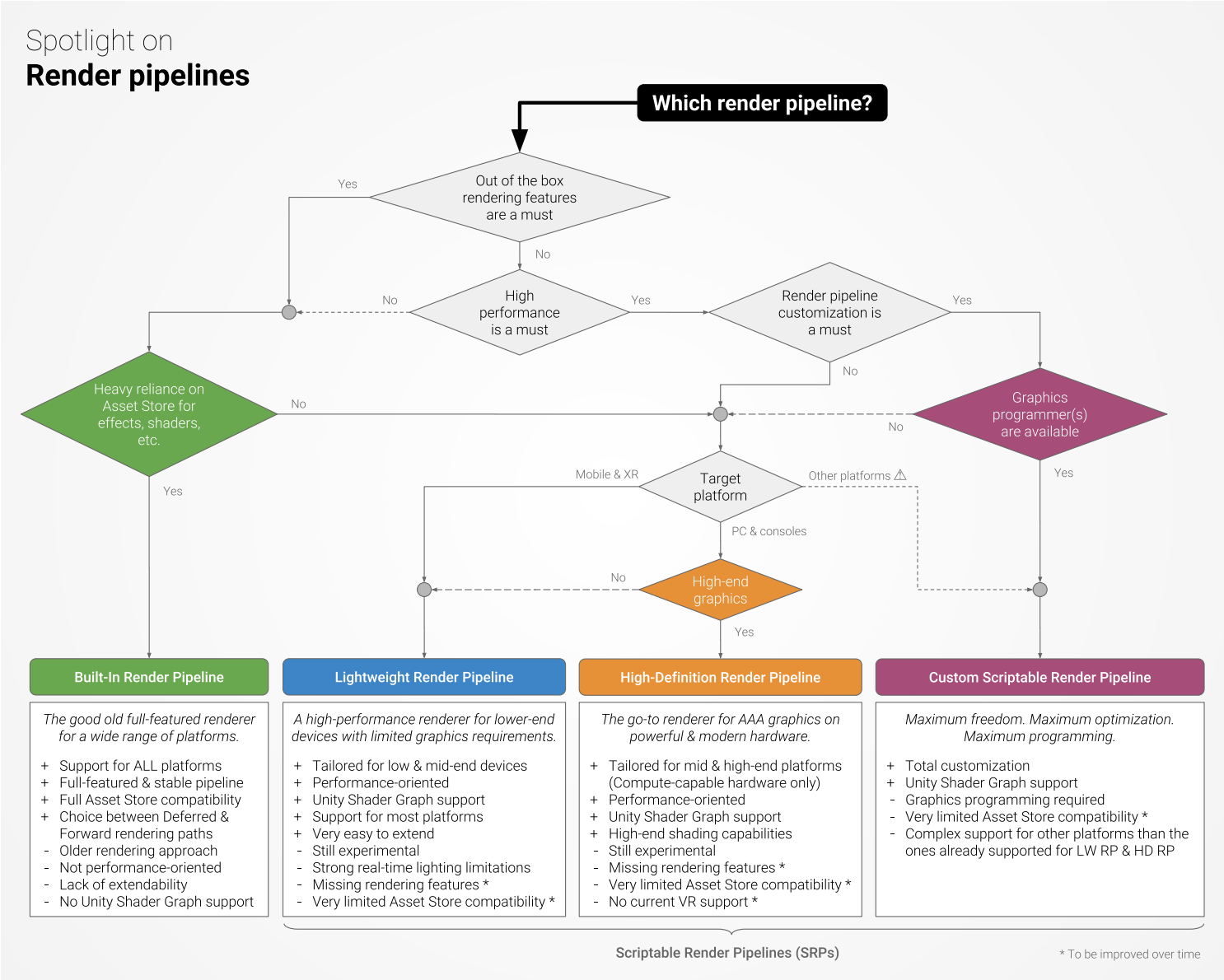
Spotlight on
Tile and Cluster Rendering



A tile is a small 2-dimensional square pixel section of the frame, and a cluster is a 3-dimensional volume inside the camera frustum. Both the tile and cluster rendering techniques rely on the listing of the lights affecting every single tile and cluster, whose lighting can then be computed in one single pass with the corresponding list of known lights. Opaque objects will most likely be shaded using the tile system, whereas transparent ones will rely on the cluster system. The main advantage this renderer offers are the faster processing of the lighting and the considerable reduction in bandwidth consumption compared to the Built-In Render Pipeline (deferred), which depends on much slower multi-pass light accumulation.

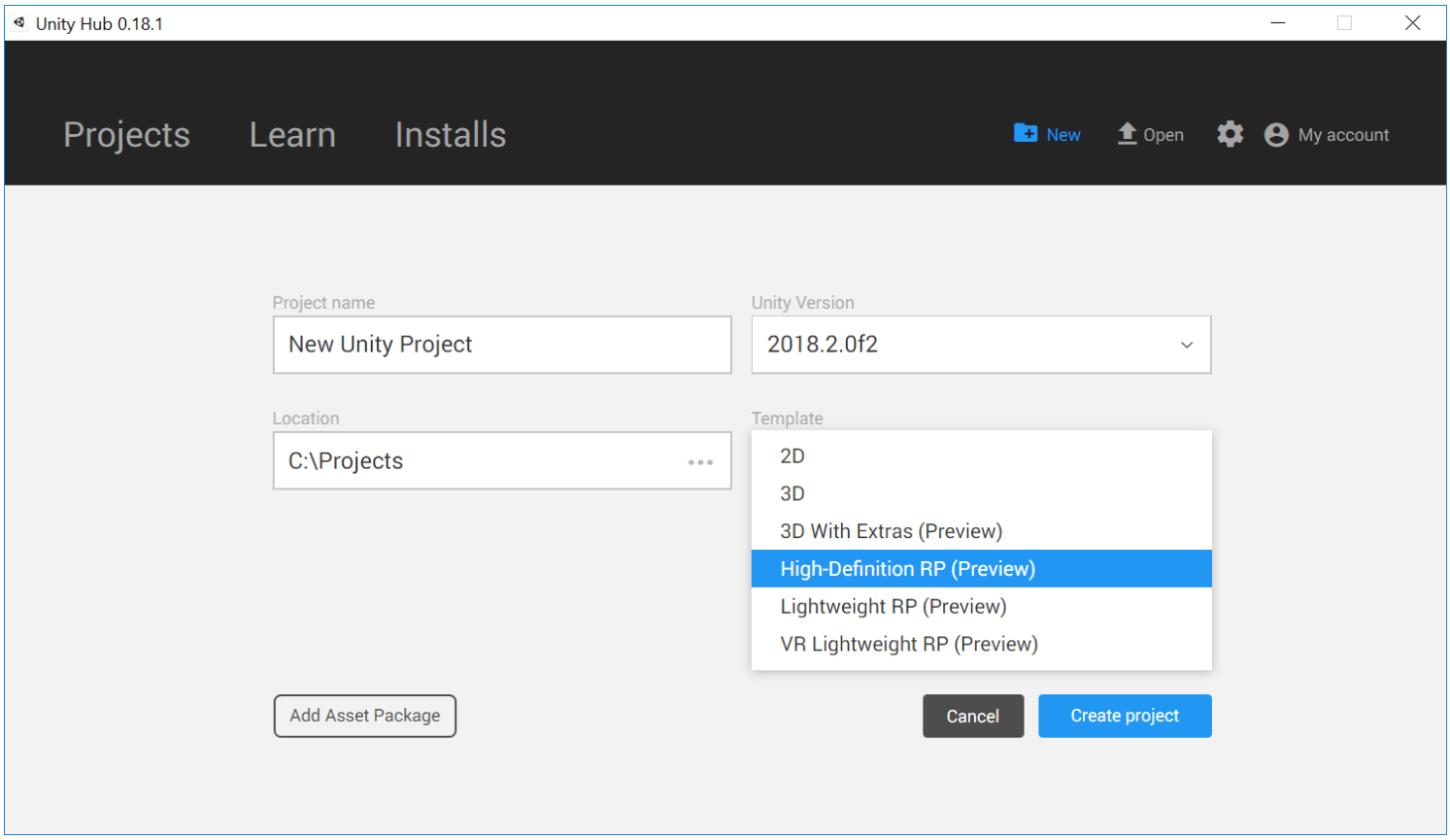
The Lightweight Render Pipeline (LWRP) is a fast single-pass forward renderer; it has been designed for devices with lower real-time lighting requirements in mind, such as smartphones, tablets and **XR** devices. The lights are culled per-object and allow for the lighting to be computed in one single pass, which results in reduced draw calls compared to the Built-In Render Pipeline.

You can now use the following decision chart to quickly find out which render pipeline you should select based on a few critical criterias.



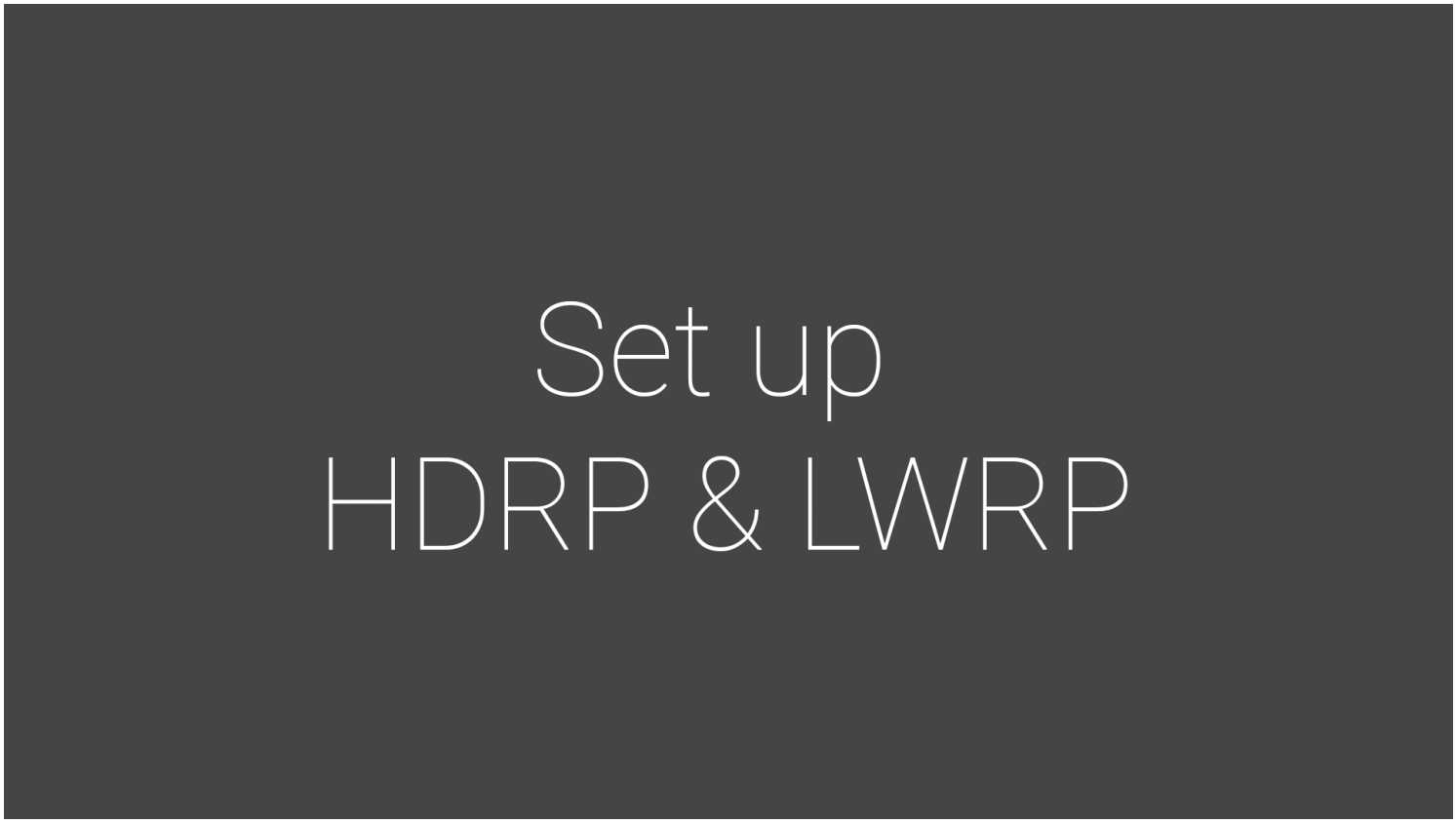
Templates

You can download the latest versions of the HDRP and LWRP via the Unity Package Manager (Window > Package Manager). The easiest way to get started with one of these SRPs is to create a new project with the Unity Hub and use one of the corresponding templates.



Manual setup

If you want to set up your project for the HDRP or LWRP by hand, ensure you have the required package installed. Then create a new asset in your Project window via Create > Rendering > High Definition Render Pipeline Asset. Drag this asset into the Graphics Settings. In case you selected the HDRP, ensure the linear color space is selected in the Player Settings and add a Rendering > Scene Settings object into your scene.



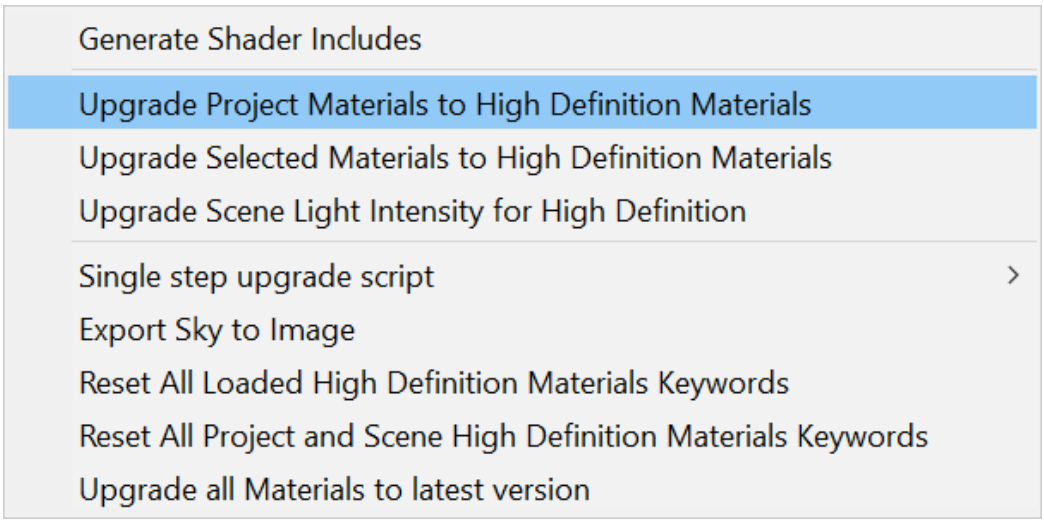
When no pipeline **asset** is assigned in the Graphics Settings window, Unity will simply use the default Built-In Render Pipeline.

Extensibility

If you have some rendering knowledge and are familiar with C#, experimenting with the SRP concept to create your own Custom Scriptable Render Pipeline is definitely recommended if you need to fully tailor the renderer for your project. The LWRP is especially easy to extend, due to its smaller shader library and the ability to inject, remove and swap rendering passes easily.

Compatibility

Porting your project's materials from the Built-In Render Pipeline to the HDRP or to the LWRP is relatively easy in Unity, thanks to a 1-click material converter under Edit > Render Pipeline > Upgrade...; be aware, however, that it is a non-reversible action. Backing up your project beforehand is highly recommended!



Nevertheless, custom shaders will have to be ported by hand, so transitioning from the Built-In Render Pipeline to the HDRP or LWRP during production might be time-consuming, depending on the number of custom shaders you would have to rewrite.

Additionally, because the High-Definition Render Pipeline is more physically correct than the Built-In Render Pipeline, especially regarding light attenuation and distribution, you should not expect your project to look identical after switching to HDRP.

Furthermore, the HDRP and the LWRP are not cross-compatible, as they do not share the same rendering features. Porting your project from HDRP to LWRP and vice versa is possible, but it is not a 1-click operation and will require manual rework of the lighting, the materials and the shaders!

Finally, the HDRP and the LWRP are still in preview and Unity is hard at work ensuring they will be production-ready very soon. Please be aware that not all features have been implemented yet for both pipelines. For instance, certain lighting modes that I detail below are not yet fully available for the LWRP, and XR is not yet properly supported by the HDRP.

Global Illumination systems

Two Global Illumination systems are available in Unity. These can be enabled in Window > Rendering > Lighting Settings.

(Precomputed) Realtime Global Illumination: This system entirely relies on Enlighten, a third-party lighting middleware. During the precomputation in Unity, Enlighten goes through two lengthy stages, among others: Clustering and Light Transport. The first one consists in simplifying the scene into a collection of surface patches called clusters, and the second, in calculating the visibility between these clusters. This precomputed data is used at runtime to generate the indirect lighting interactively. The strength of Enlighten relies the ability to edit the lighting in realtime, as the precomputed data relies on the relation between clusters. However, like in other traditional lightmapping techniques, editing the static geometries in your scene will trigger a new precomputation.

Baked Global Illumination: The lighting is baked into textures called lightmaps, and into Light Probes. The Baked GI system can use one of the following lightmappers:

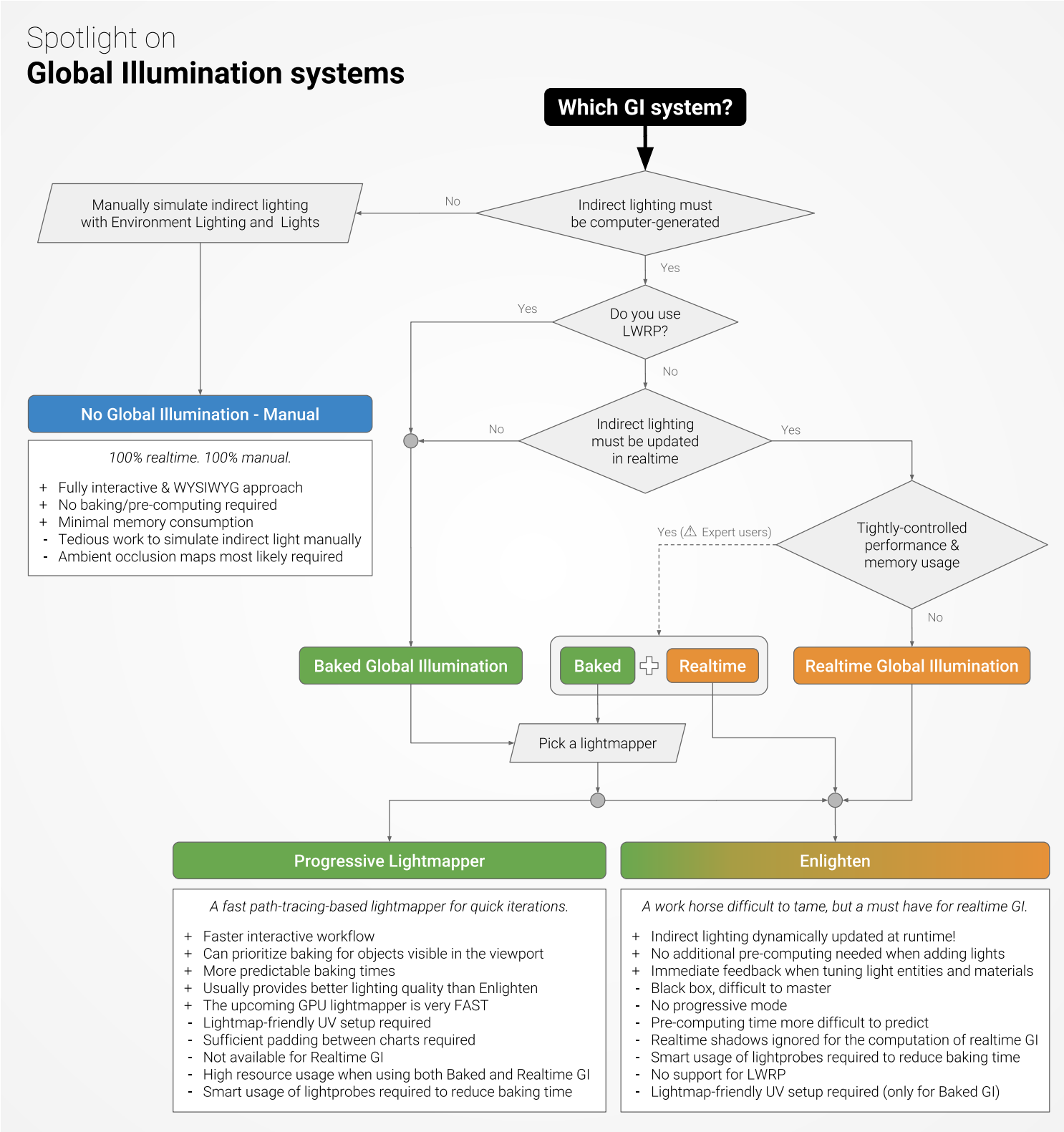
Progressive Lightmapper

Enlighten

The Progressive Lightmapper can prioritize the computation of the lighting for objects visible to the camera and greatly speed up the iteration on the lighting, at the cost of increasing the overall baking time for the entire scene. The Progressive Lightmapper uses the CPU to calculate the indirect lighting using path tracing. A new GPU Progressive Lightmapper is currently in development and will radically reduce the baking time for your scenes.

Because both Enlighten and the Progress Lightmapper use different methods to produce the baked lighting, you should not expect the resulting lighting to match exactly when comparing them.

Have a look at the diagram below to decide which Global Illumination system is recommended for your project, as well as its main advantages and disadvantages



Static versus Dynamic

No matter which Global Illumination system you use, Unity will only consider objects that are marked as “Lightmap Static” during the baking/precomputing of the lighting. Dynamic (i.e. non-static) objects have to rely on the Light Probes you placed throughout the scene to receive indirect lighting.

Because the baking/precomputing of the lighting is a relatively slow process, only large and complex assets with distinct lighting variations, such as concavity and self-shadowing, should be tagged as “Lightmap Static”. Smaller and convex meshes that receive homogeneous lighting should not be marked as static, and they should therefore receive indirect lighting from the Light Probes which store a simpler approximation of the lighting. Larger dynamic objects can rely on LPPVs, in order to receive better localized indirect lighting. Limiting the number of objects tagged as “Lightmap Static” in your scene is absolutely crucial to minimize baking

times while maintaining an adequate lighting quality. You can learn more about this optimization process and the importance of Probe lighting in this tutorial.

Warning

Unity allows both the Baked and Realtime GI systems to be active simultaneously, which gives you access to all lighting features. However, you must be warned that enabling both systems greatly increases the baking time and the memory usage at runtime because these systems do not rely on the same data sets. Furthermore, the interactive update of the indirect lighting at runtime will put additional strain on the CPU, and you can expect discrepancies when visually comparing the indirect lighting provided by the Baked and the Realtime GI systems, as they rely on different techniques to simulate the indirect lighting and often operate at significantly different resolutions.

You should restrict the usage of both GI systems to high-end platforms and/or to projects that have tightly controlled scenes with predictable costs. This approach should only be used by expert users who have a very good understanding of all lighting settings because managing both systems adds great complexity. Consequently, picking one of the two GI systems is usually a safer strategy for most projects. Using both systems is rarely recommended!

Light modes

The mode of a light is a property that is a common source of confusion. Most importantly, the mode of a light is only relevant if the Baked Global Illumination system is enabled.

There are three modes available in the Light **Inspector**:

Baked: The direct and indirect lighting from these lights are baked into lightmaps, which can be a time-consuming process. There is no runtime cost to process these lights, however applying the resulting lightmaps to the scene does have a minor cost.
Realtime: The direct lighting and shadows from these lights are real-time and therefore not baked into lightmaps. Their runtime cost can be high, depending on the complexity of the scene, the number of shadow casting lights, the number of overlapping lights, **etc**. Furthermore, if you enable Realtime Global Illumination, further performance loss will incur in order to update the indirect lighting at runtime.
Mixed: This is a hybrid mode that offers a mix of baked and real-time features, such as baked indirect lighting and real-time direct lighting. The behavior of all Mixed lights in your scene and their performance impact will depend on the chosen global mixed Lighting Mode that I describe in the next section.
If you do not use any GI system or only use the Realtime GI system, then all Baked and Mixed lights will be overridden to Realtime!

Spotlight on
Light mode availability

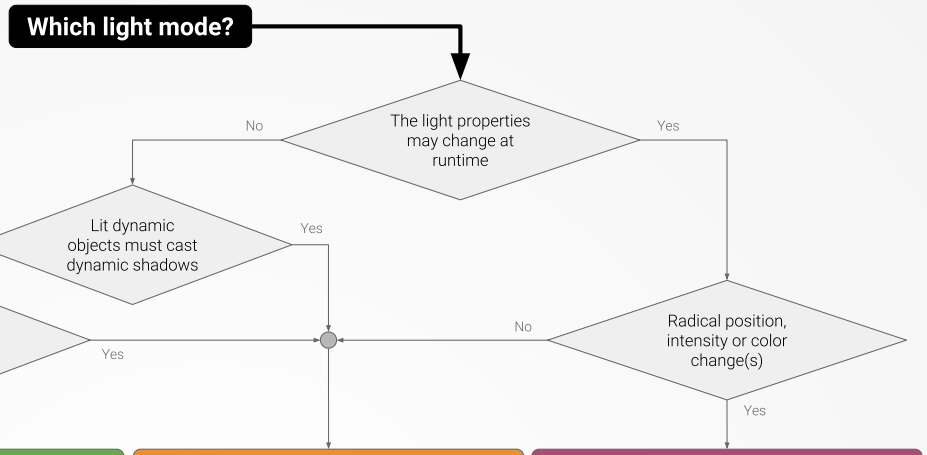
	Baked light mode	Mixed light mode	Realtime light mode
Baked Global Illumination	✔	✔	✔
Baked & Realtime Global Illumination	✔	✔	✔
Realtime Global Illumination	✘	✘	✔
No Global Illumination	✘	✘	✔

The following diagram combines a decision flowchart with a comparison table; it can help you decide which **light mode** is appropriate every time a new light is added into the scene.

Spotlight on

Light modes

When using Baked GI, or Baked & Realtime GI



	Baked	Mixed	Realtime
	A cheap precomputed light. + No runtime cost + Lit static objects produce indirect light + Lit static objects produce soft baked shadows - Cannot change light properties at runtime - Lit surfaces do not receive specular highlights - Light quality depends on lightmap resolution - Light baking/pre-computing is slow **Typical use case:** Background and ambient light on higher end devices. Most lights for low-end devices.	*Maximum flexibility.* + Lit dynamic objects can cast dynamic shadows + Lit static objects produce indirect light + Lit surfaces can receive specular highlights - Higher runtime cost - Light baking & pre-computing is slow **Typical use case:** Fixed sunlight, fixed gameplay light, shadow casting key light.	*The go-to light mode for fully interactive lighting.* + Instant update when changing light properties + Lit dynamic objects cast dynamic shadows + Lit surfaces receive specular highlights + No pre-computing required when adding Lights - Higher runtime cost - No indirect light when using Baked GI only **Typical use case:** Moving sunlight, destructible or flickering light, gameplay light.
Runtime cost	Minimal	Higher	Highest
Runtime update (position, intensity, etc.)	✘	✔ ⚠ Disconnection between baked & direct lighting	✔
Objects receive dynamic specular highlights	✘	✘ Subtractive ✔ Baked indirect ✔ Shadowmask mode\Shadowmask ✔ Shadowmask mode\Distance Shadowmask	✔
Contributes to Baked GI	✔*	✔*	✘
Contributes to Realtime GI	✔*	✔*	✔*
Static objects cast realtime shadows	✘	✘ Subtractive ✔ Baked indirect ✘ Shadowmask mode\Shadowmask (✔ HDRP) ✔ Shadowmask mode\Distance Shadowmask	✔
Dynamic objects cast realtime shadows	✘	✔ ⚠ No realtime shadow beyond *Shadow Distance*	✔ ⚠ No realtime shadow beyond *Shadow Distance*
Iteration time	Slowest, due to baking	Slow	Fast

⚠ Overridden to Realtime mode when using only Realtime GI or using no GI system at all

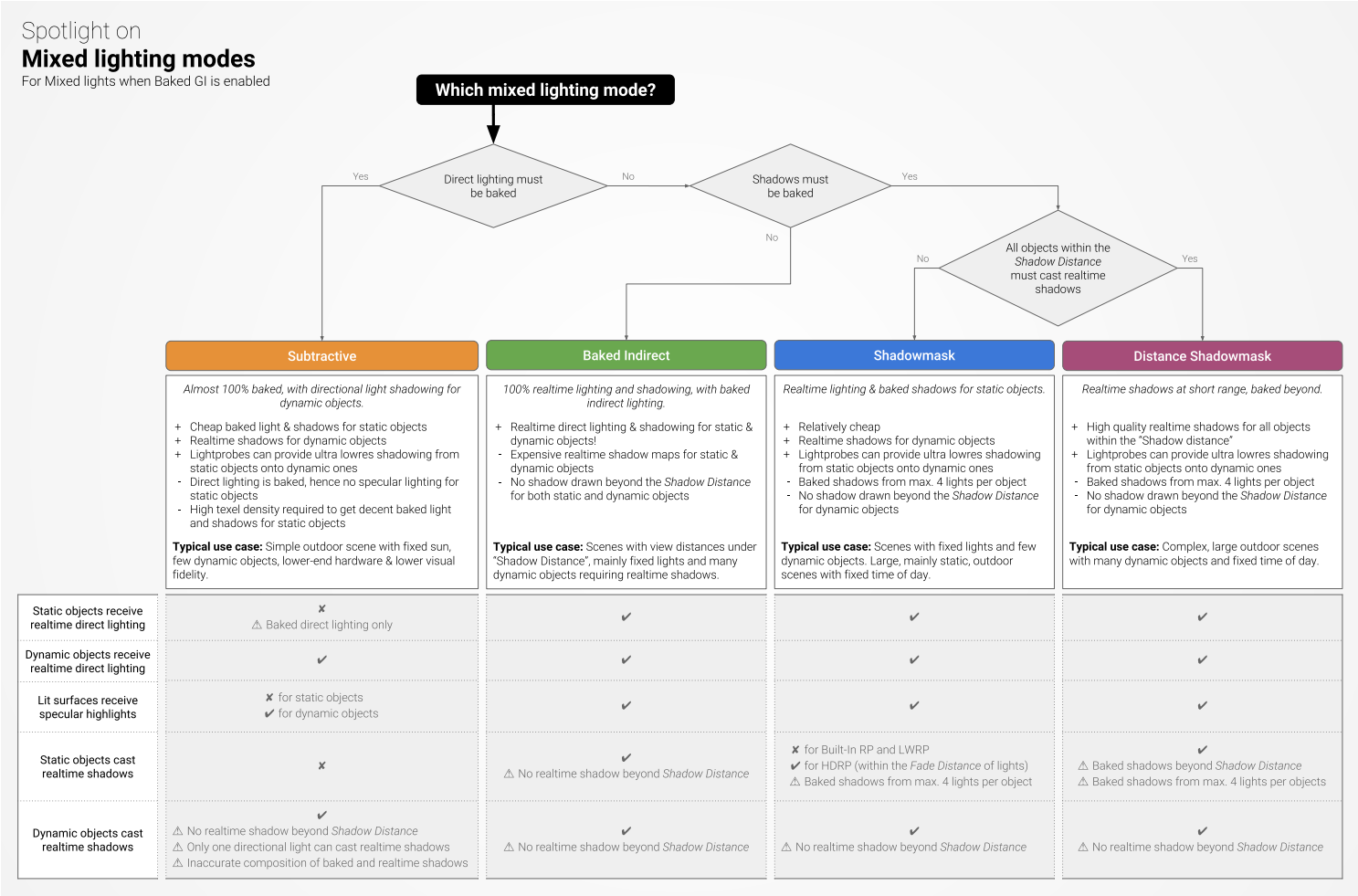
* Only if lit objects are marked as "Lightmap Static"

Mixed lighting modes

As you can see in the previous diagram, Mixed lights have specific baked and real-time capabilities, depending on the global mixed Lighting Mode that you picked in the Lighting > Settings window.

There are four modes to choose from:

Subtractive
Baked Indirect
Shadowmask Mode: Shadowmask
Shadowmask Mode: Distance Shadowmask



The Shadowmask mode and the Shadow Distance can be tuned under Edit > Project Settings > Quality. When using the HDRP, the Shadowmask mode is enabled in the HDRenderPipelineAsset assigned in the Graphics settings, whereas the shadow Max Distance is set in the Scene Settings object.

The HDRP supports a new hybrid Shadowmask Mode. You can control whether a specific light should cast real-time shadows via the Non Lightmapped Only checkbox in Additional Settings. If this parameter is used, then the light will cast real-time dynamic shadows when the camera is within the Fade Distance of the light, otherwise it will fall back to the baked Shadowmask. The main advantage of this new mode for HDRP is the ability to use the baked Shadowmask for specific lights that are within the Shadow Distance used by the main directional light, instead of real-time shadows.

Pipeline comparison table

The LWRP and HDRP are still in preview, which means that on the one hand they bring exciting new features to the table, but on the other hand, they might not support certain functions offered by the Built-In Pipeline yet, or drop support for others. The following table gives you an overview of the current state of the lighting pipeline for Unity 2018.3.

Render Pipeline Comparison
2018.3

	Built-In Render Pipeline	Lightweight Render Pipeline	High-Definition Render Pipeline
Platform Coverage	• All platforms	• Most platforms	• DX11, Playstation 4, Xbox One, Metal & Vulkan • XR unsupported yet ⚠
Rendering Paths	• Multi-pass Forward • Multi-pass Deferred	• Single-pass Forward • No support for Deferred yet	• Cluster/Tiled Deferred • Cluster/Tiled Forward
Color Space Mode	• Linear with sRGB light intensity • sRGB	• Linear with linear light intensity • Gamma (for legacy hardware)	• Linear with linear light intensity

Light entities	• Non-physically based intensity unit
	• Non-physically based falloff
	• Realtime Directional, Spot and Point lights
	• Max # lights in forward per pixel: 8
	• Max # lights in deferred per pixel: unlimited
	• Max # lights per vertex: 4
		• Non-physically based intensity unit
		• Non-physically based falloff
		• Realtime Directional, Spot and Point lights
		• Baked Rectangle Area light
		• Max # lights: 1 directional + 8 light
		• Max # lights per vertex: 4
			• Physically based intensity units
			• Physically based falloff
			• Realtime Directional, Spot and Point lights
			• Spot light types
			  ○ Cone
			  ○ Pyramid
			  ○ Box
			• Realtime Rectangle Area light
			• Realtime Line light
			• Max # lights per pixel: 24

Realtime Shadow Casting Lights	• Directional • Spot • Point	• Directional • Spot	• Directional • Spot • Point

Light Modes	• Baked
	• Mixed
	  ○ Subtractive
	  ○ Baked Indirect
	  ○ Shadowmask Mode: Shadowmask
	  ○ Shadowmask Mode: Distance Shadowmask
	• Realtime
		• Baked
		• Mixed
		  ○ Subtractive
		• Realtime
			• Baked
			• Mixed
			  ○ Baked Indirect
			  ○ Shadowmask (hybrid mode)
			• Realtime

Light Probe Modes	• One interpolated probe • LPPV • Occlusion probe • LPPV Occlusion probe	• One interpolated probe	• One interpolated probe • LPPV • Occlusion probe • LPPV Occlusion probe

Shader Library	• Unified PBS Shader
	  ○ Metallic workflow
	  ○ Specular workflow
	• Many non-PBS shaders
		• Unified PBS Shader
		  ○ Metallic workflow
		  ○ Specular workflow
		• Unified non-PBS Shader
		  ○ Covers all non-PBS shaders from Built-in RP
			• Advanced Unified PBS Shaders
			  ○ Metallic workflow
			  ○ Specular workflow
			• Supported features
			  ○ Subsurface Scattering
			  ○ Anisotropic
			  ○ Iridescence
			  ○ Clear coat
			  ○ Tessellation
			  ○ Parallax Occlusion Mapping
			  ○ Detail map
			  ○ Support for UV0 to UV3
			  ○ Planar and Tri-Planar projections
			  ○ Layering
			  ○ Decal
			  ○ And more....

Global Illumination	• Baked Directional, Spot, Point and Rectangle Area Lights • Baked Lightmap (Non-Directional and Directional) • Baked Light Probes • Realtime Light Probes • Realtime GI	• Baked Directional, Spot, Point and Rectangle Area Lights • Baked Lightmap (Non-Directional and Directional) • Baked Light Probes • Realtime GI Not Supported ⚠	• Baked Directional, Spot, Point and Rectangle Area Lights (Line light not supported yet) • Baked Lightmap (Non-Directional and Directional) • Baked Light Probes • Realtime Dynamic Lightmap • Realtime Light Probes • GI with inverse square falloff
Anti-Aliasing	• MSAA • FXAA • TAA	• MSAA • FXAA	• TAA • MSAA in progress for forward
Motion Vectors	• Motion Vectors	• None	• Motion Vectors • Distortion Vectors

Post-Processing	• Legacy Post-Processing stack
	• Post-Processing Stack V2
		• Post-Processing Stack V2, except for
		  ○ TAA
		  ○ Motion Blur
		  ○ SSR
		  ○ HDR color grading
			• Post-Processing Stack V2, except for
			  ○ SSR

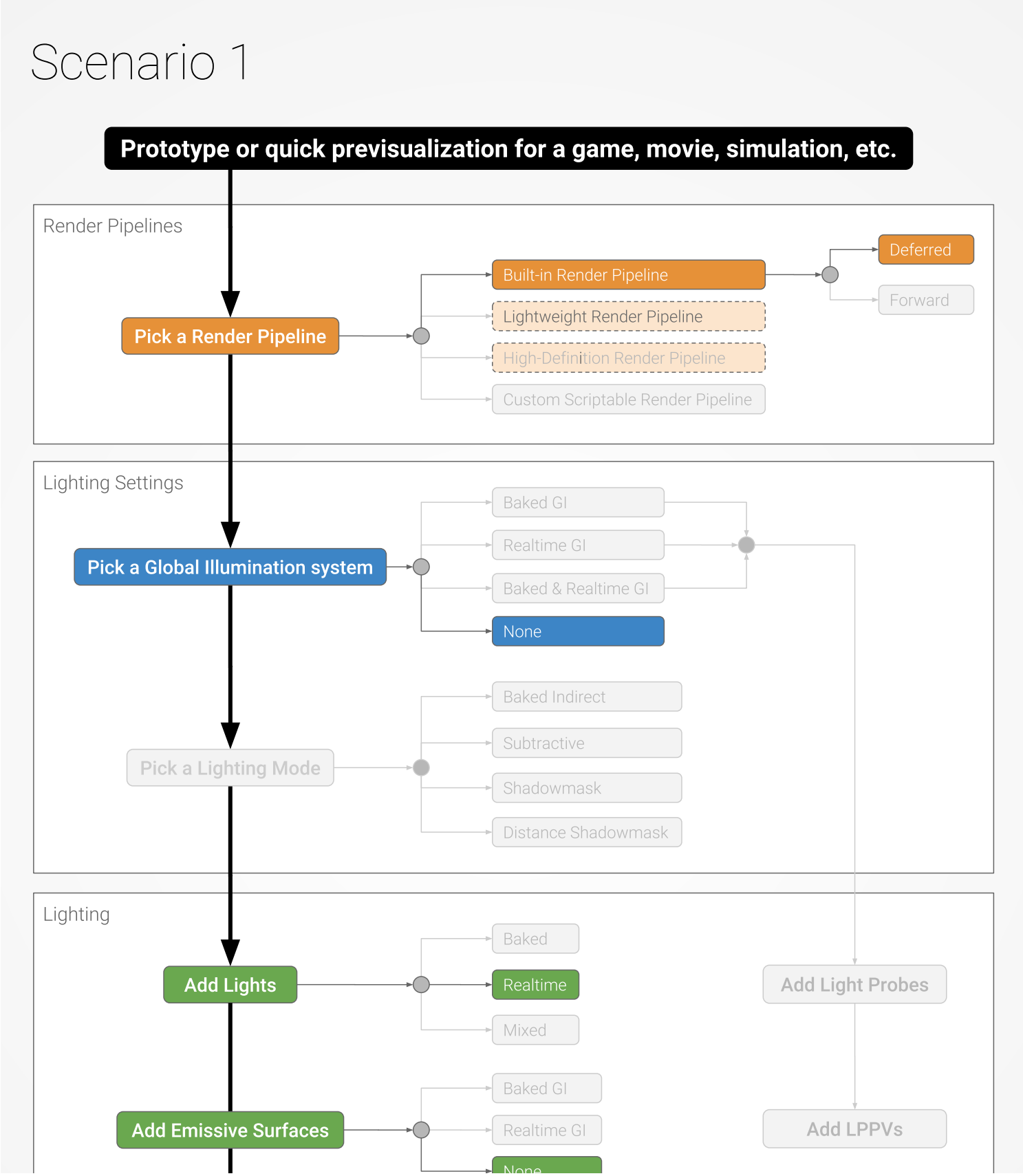
Sky lighting	• Procedural Sky • Cubemap/LatLong Sky • Ambient Lighting	• Procedural Sky • Cubemap • Ambient Lighting	• Dedicated sky manager system, no ambient light • GGX GPU convolution of the sky • Procedural sky (from Built-In RP) • HDRI Sky

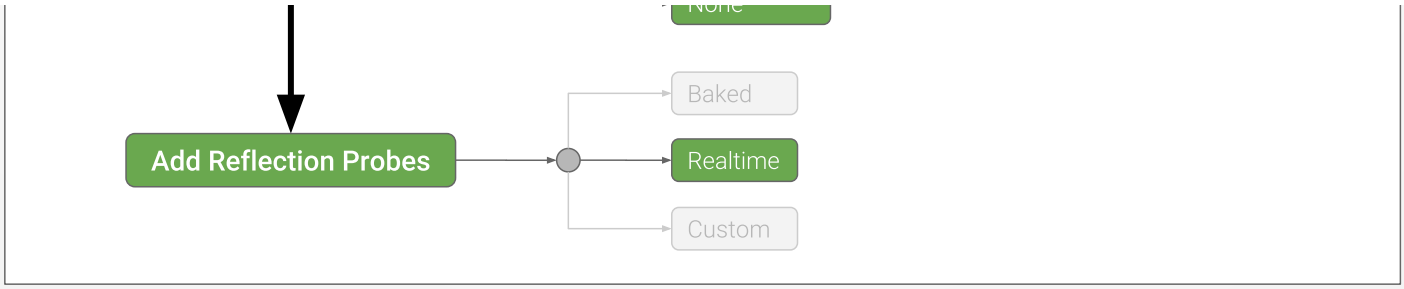
Unsupported		• Projector
		• GrabPass (replaced by OpaqueTexture)
		• Realtime GI
		• Light Cookie
		• Mixed lighting modes
		  ○ Baked Indirect
		  ○ Shadowmask & Distance Shadowmask
		• Transparent Shadows
			• Terrain material
			• Projector material
			• Gamma color space
			• XR

Lighting scenarios

Now that we have introduced the render pipelines and the main lighting features, let's have a look at a few examples of projects and see which settings could be used to light them. Since every project is unique, you might use slightly different options based on your requirements.

1. Prototype or quick previsualization

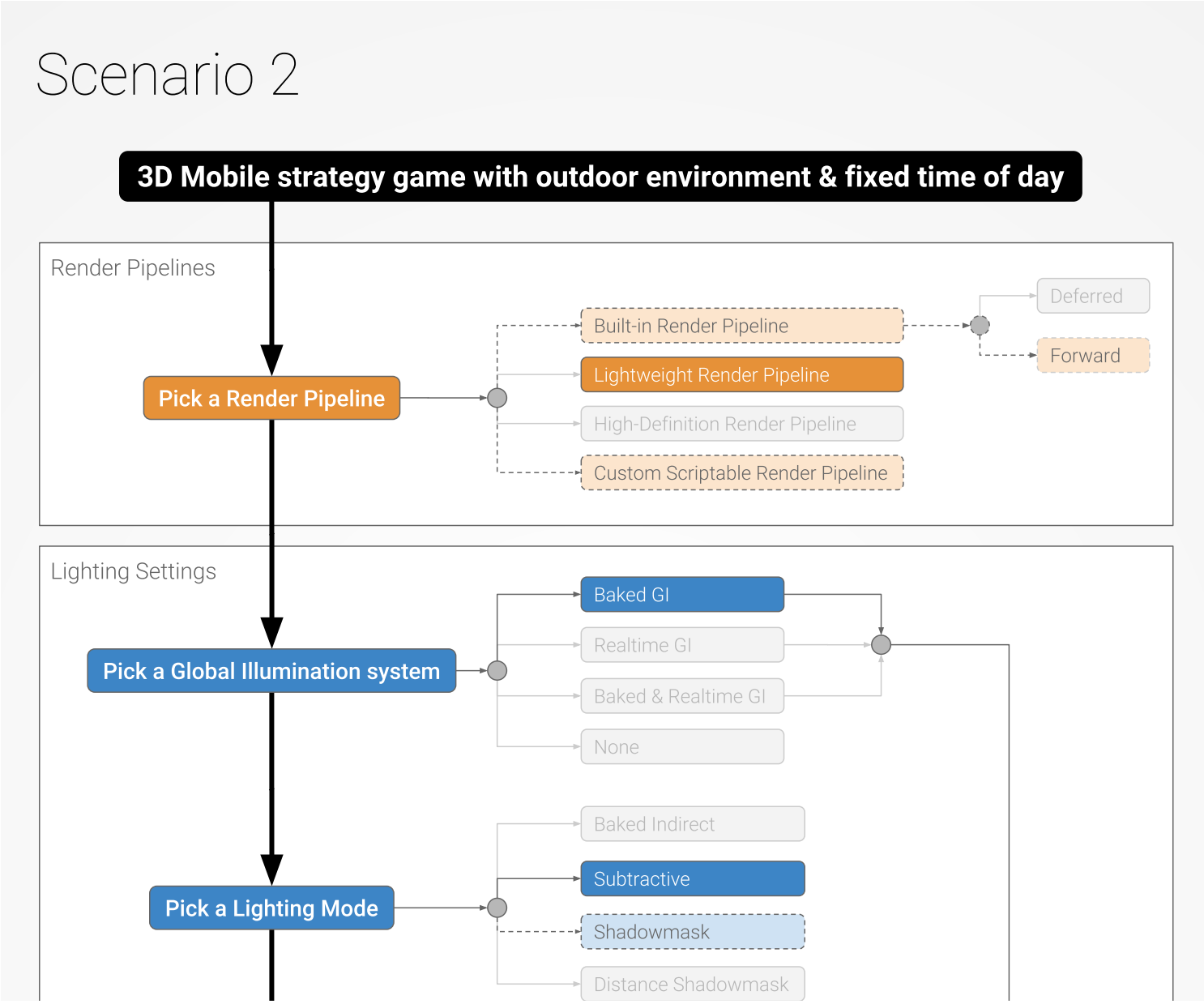


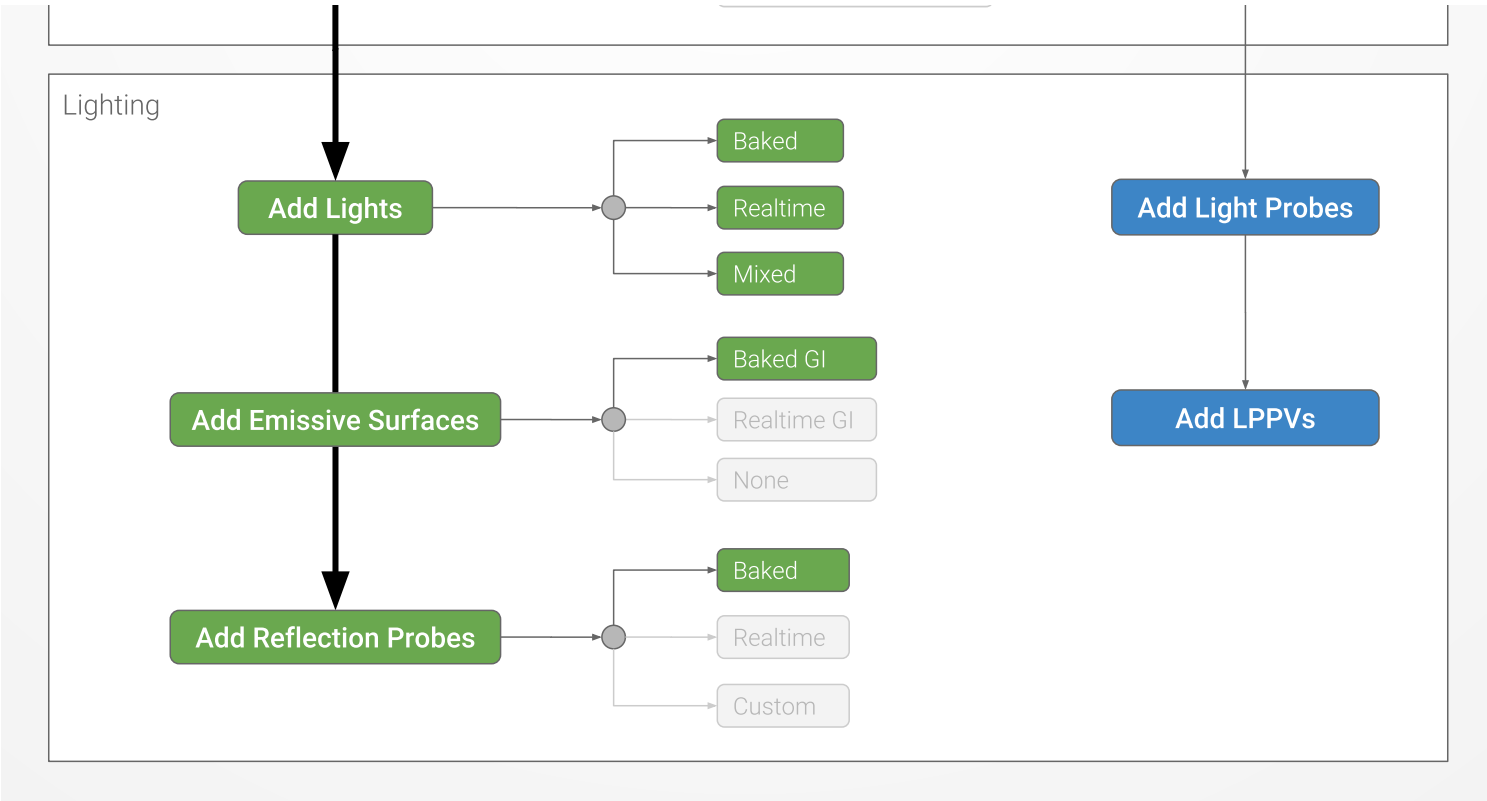


If you heavily rely on the **Asset Store** to build your prototype, the Built-In Render Pipeline could be the only suitable render pipeline, as most assets found on the Store are not fully compatible with the HDRP and LWRP; nonetheless, asset compatibility will improve over time. If you are building all the assets from the ground up and already have a clear idea of your project's requirements, then you could pick one of the two SRPs (i.e. LWRP or HDRP) or create a custom one.

When you are in the early stage of (pre-)production and need a quick turnaround and maximum flexibility for the lighting, you might prefer a full real-time approach that does not require any precomputation, therefore you might want to turn off both the Baked and Realtime GI systems. To alleviate the lack of proper indirect lighting, you can enable Screen Space **Ambient Occlusion** from the Post Processing Stack V2: it can help ground the object in the scene by offering cheap real-time contact shadows.

2. 3D Mobile strategy game



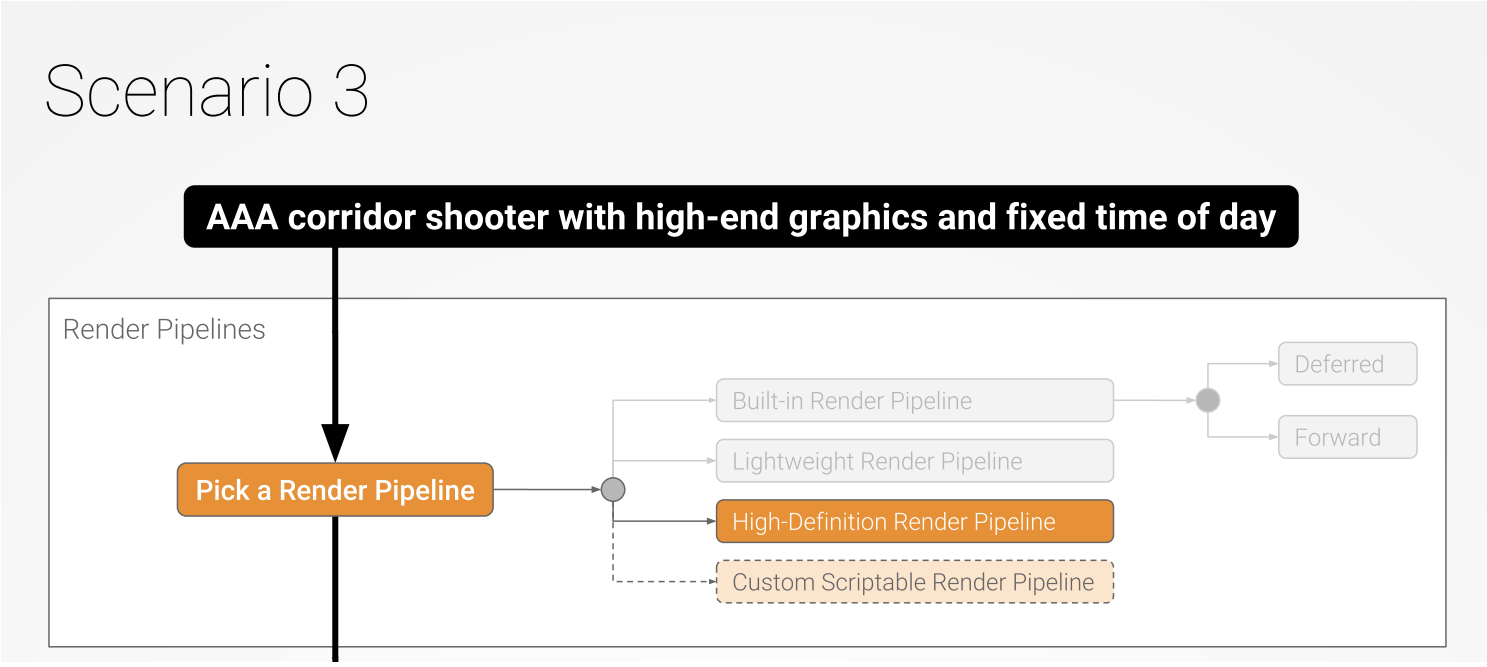


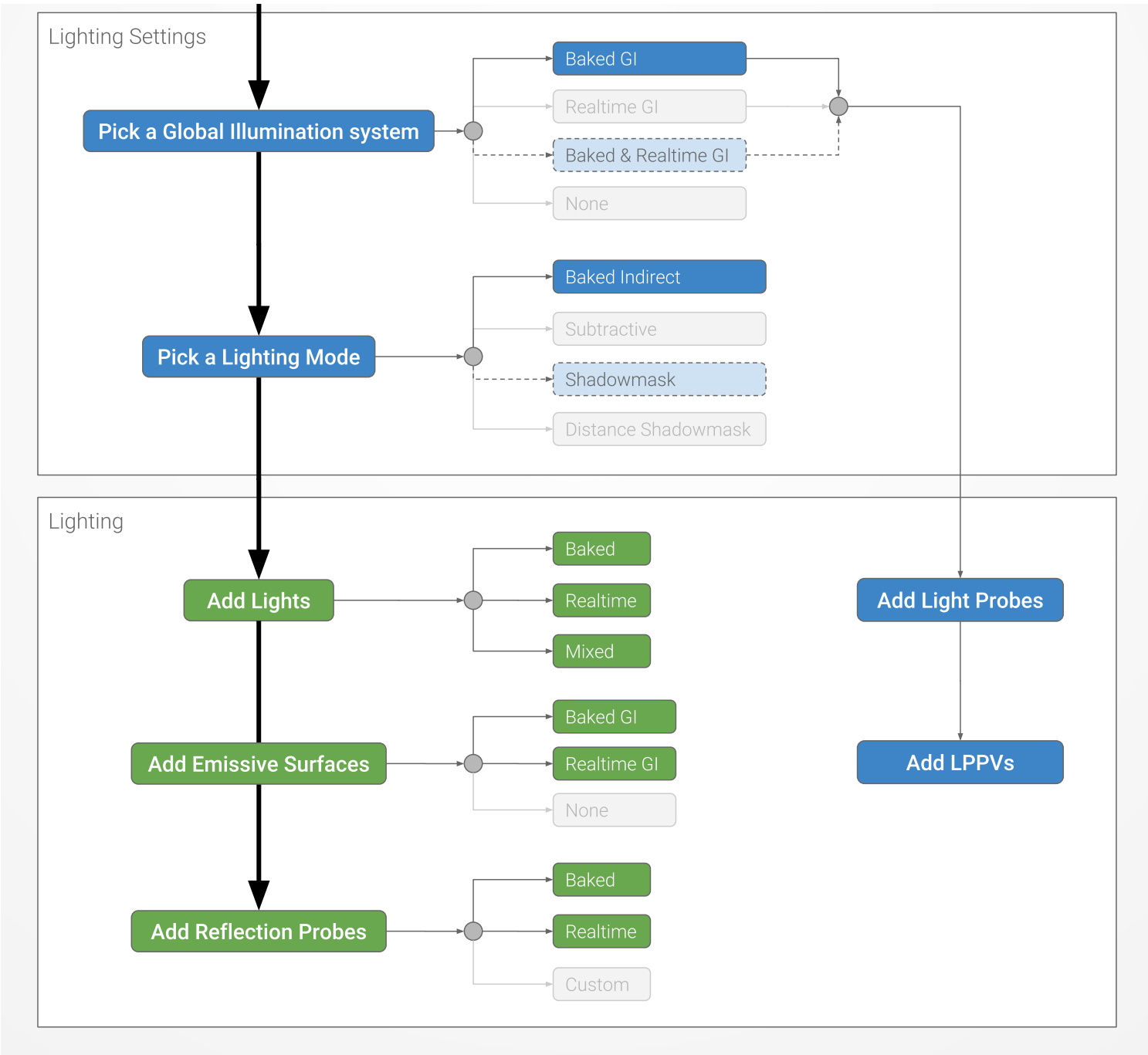
If you are targeting mobile devices, the LWRP could be a great candidate to ensure solid performance for your strategy game. If the rendering pipeline needs to be customized to better suit your game, a graphics programmer will probably find extending the LWRP straightforward.

If you pick the LWRP and use Baked Global Illumination, be aware that at the moment, only the Subtractive Lighting Mode is available for the Mixed lights. Support for Baked Indirect and Shadowmask will be added in a later release.

Alternatively, if you decide to stick to the older Built-In Render Pipeline because, for example, you rely on many assets from the Asset Store, all global mixed lighting modes are supported. In this case, an approach with the Shadowmask Lighting Mode will provide baked shadows while still allowing dynamic objects to cast real-time shadows. If Shadowmasks are too expensive for your project, you can fall back to the cheapest Subtractive mode. Finally, the forward **rendering path** is probably the best option if you have a very small number of lights in your level(s), and if you're targeting older hardware.

3. AAA corridor shooter (fixed time of day)





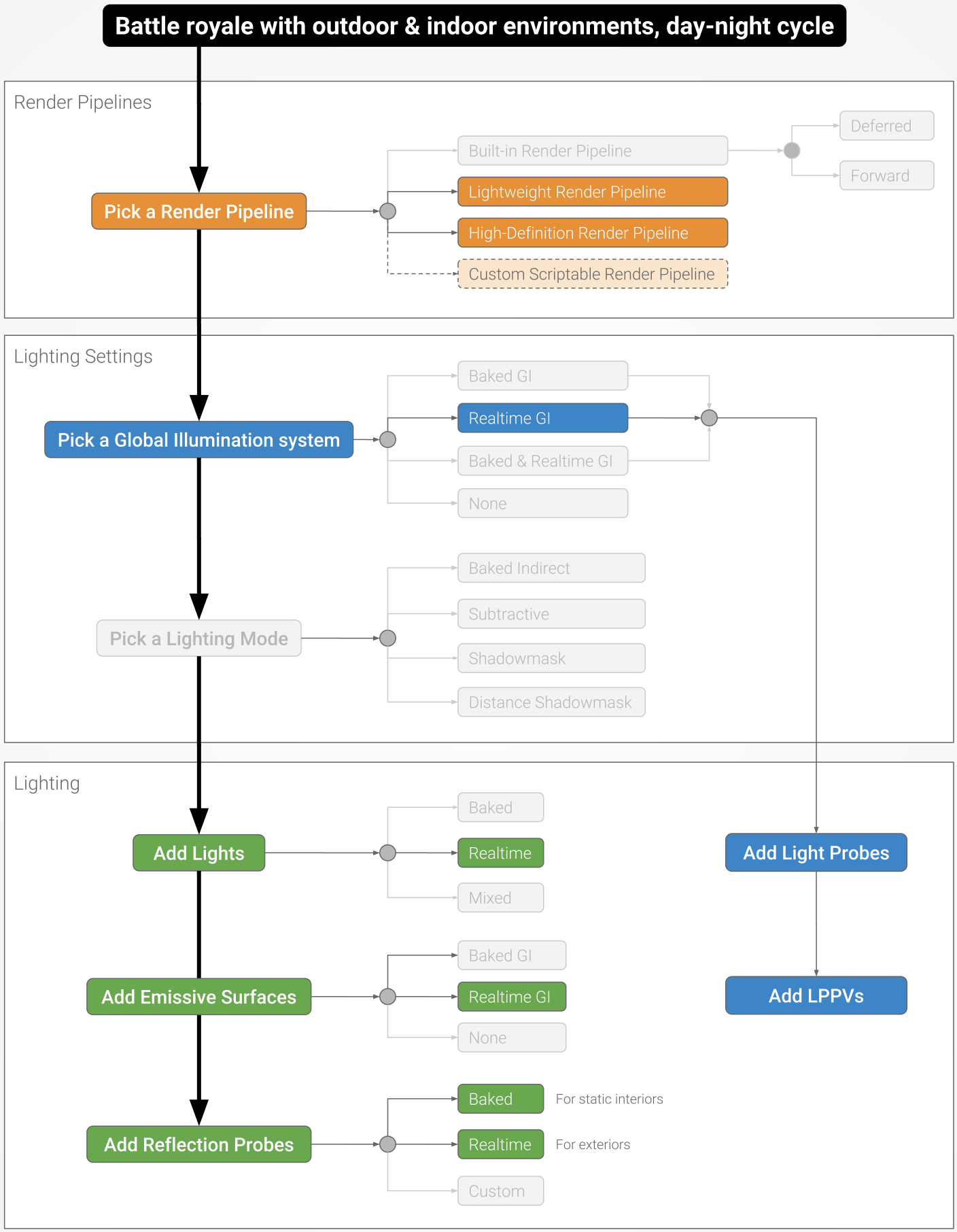
If you are aiming for AAA-quality visuals on PC and consoles for your linear first-person shooter, the HDRP should be the preferred render pipeline. Again, with the help of graphics programmers, a custom SRP could also be developed.

If your levels contain many real-time shadow casting lights (e.g. destructible light props and moving lights), then using the Baked GI system with the Baked Indirect mode should ensure you get great looking indirect lighting from the Mixed directional light and the **Baked lights** in static light props. If your levels consist of a larger proportions of fixed shadow casting lights, then an approach with Shadowmasks could be recommended because the HDRP offers a great hybrid Shadowmask mode which gives you more control over the blend between real-time and baked shadows.

Because this type of linear game is generally highly-predictable in terms of performance and memory consumption, both the Baked and Realtime GI systems could be activated simultaneously. However, as explained in the Global Illumination section, using both systems concurrently will significantly increase the performance cost and the baking time, and it should only be used by expert users who understand all the technical implications!

4. Battle royale (day-night cycle)

Scenario 4


Battle royale with outdoor & indoor environments, day-night cycle
Render Pipelines
Pick a Render Pipeline
Built-in Render Pipeline
Lightweight Render Pipeline
High-Definition Render Pipeline
Custom Scriptable Render Pipeline
Deferred
Forward
Lighting Settings
Pick a Global Illumination system
Baked GI
Realtime GI
Baked & Realtime GI
None
Pick a Lighting Mode
Baked Indirect
Subtractive
Shadowmask
Distance Shadowmask
Lighting
Add Lights
Baked
Realtime
Mixed
Add Emissive Surfaces
Baked GI
Realtime GI
None
Add Reflection Probes
Baked
For static interiors
Realtime
For exteriors
Custom
Add Light Probes
Add LPPVs

If you plan to release a battle royale game for PC and consoles, that features large-scale environments and fully dynamic lighting, you should select the HDRP, or extend it to tailor the rendering pipeline to your project. You could consider the LWRP if you are not aiming for AAA visual fidelity and are targeting mobile devices or systems with lower specifications.

In order to accommodate for the day-night cycle, if you selected the HDRP, the Realtime GI system should be activated to simulate the indirect lighting at any time of day. To maximize performance in certain densely lit interiors, you could set the Indirect Multiplier of certain lights at 0, if you want the Realtime GI system to ignore these and minimize their rendering cost.

The LWRP does not support the Realtime Global Illumination system: the day-night cycle would have to be handled with a custom script that would, for instance, modulate the sun and ambient color throughout the day.

For this particular scenario, activating both the Realtime GI and the Baked GI systems is not recommended, because the resulting overhead in terms of performance and scene management for an immense level could be critical. Another argument against the use of both GI systems is the unpredictable nature of such large-scale multiplayer game: performance estimations are for instance more difficult than in a highly-scripted single player adventure.

Final words

The rendering landscape has changed radically in Unity over the past months thanks to the introduction of the Scriptable Render Pipelines. Therefore, keeping up with all these changes and their implications for the lighting pipeline can be exhausting.

Hopefully, this guide and its many illustrations have given you a better understanding of the capabilities of each render pipeline so that you can confidently light your projects in Unity with the appropriate lighting settings!

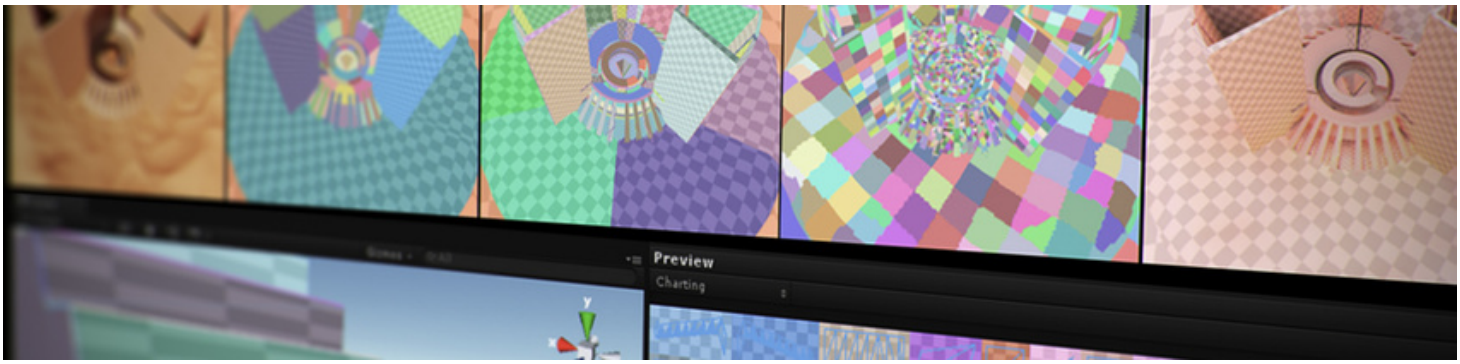
You can learn more about the lighting in Unity and the rendering pipelines with the following articles:

Introduction to Lighting and Rendering tutorial
Use the Unity Hub to get started with HDRP, LWRP or Built-In RP
Creating Believable Visuals tutorial (Built-In RP)
Lightweight Render Pipeline on GitHub
High-Definition Render Pipeline on GitHub

2018–08–17Page published with no editorial review

Expert guides

Leave feedback



The Expert Guides in this section are written by Unity's area experts in their own words. Here, Unity developers provide detailed insights into features and topics relevant to using Unity and how to get the most out of it.

Some of these documents were blog posts written specificially by developers who want to get their knowledge into your hands directly. Because their length and format differs from the the User Manual's normal style, we provide them to you as downloadable PDFs or Google documents.

Guides

Windows Performance Analysis
Author: Tautvydas Zilys, Developer - Windows Team
Date: 2016–06–15
Synopsis: This extensive set of guides from Tautvydas explains how to use *Event Tracing for Windows* to analyze your Windows-based games, including both standalone players and **Universal Windows Platform** players running on all PCs and devices, as well as in the Unity Editor.

Particle System Modules FAQ
Author: Karl Jones, Developer - Sustained Engineering
Date: 2016–04–20
Synopsis: Karl talks through scripting access to the **particle system** modules, and explains what goes on under the hood when you access and edit **particle system** properties in your code. *(originally posted on the Unity blog)*

Realtime GI on desktops and consoles
Author: Jesper Mortensen, Graphics Engineer - GI & Lighting Lead
Date: 2015–11–05
Checked with Unity version: 5
Synopsis: Jesper goes into detail about the techniques used to produce *The Courtyard*, a demo that puts the precomputed realtime global illumination (GI) features in Unity 5 to good use. *(originally posted on the Unity blog)*

Monitor Calibration
Author: Laurent Harduin, Senior Lighting Artist
Date: 2016–10–28
Checked with Unity version: 5.5
Synopsis: Monitor Calibration ensures you get true and consistent colors and luminance on your

monitor. It is an often overlooked development practice that is essential when working with a team of artists or developers. This guide takes you through the process of calibrating your monitor step-by-step.

Using Enlighten with Unity
Author: Kasper Engelstoft, Developer - Lighting Team
Date: 2017–01–17
Checked with Unity version: 5.5
Synopsis: This guide details how Unity uses **Enlighten** for computing global illumination (GI). Kasper Englestoft explains the workflow and the equations Unity uses to calculate lighting, and the benefits of using it.

High Dynamic Range Imaging (HDRI)
Author: Sébastien Lagarde, Developer - Rendering Research
Date: 2017–03–09
Checked with Unity version: 5.5
Synopsis: This extensive guide details many aspects of capturing and using **HDRIs**. Sébastien Lagarde explains the implementation and calibration of **HDRIs** in Unity, to help you achieve accurate real world environmental lighting.

Unity Photogrammetry Workflow
Author: Sébastien Lagarde, Developer - Rendering Research
Date: 2017–06–23
Checked with Unity version: 2017.1
Synopsis: This document is aimed at game studios. It describes a photogrammetry workflow dedicated to authoring game **assets** on an affordable budget. While it is possible to create extremely high quality **assets** for non-game use using photogrammetry, this documents focuses on using photogrammetry for game development. It describes how to get the best results within the typical time and budget constraints of game development.

De-Lighting in Unity
Author: Sébastien Lagarde, Developer - Rendering Research
Date: 2017–10–20
Checked with Unity version: 2017.1
Synopsis: Photogrammetry is becoming more and more popular in the CG industry. Being able to use a regular **camera** as a powerful 3D scanner is now an essential tool when creating realistic 3D assets. However, when you generate raw textures in this way, they often contain a lot of lighting information that you should remove to obtain the true base color for **rendering**. This document shows you how to solve this problem using the Unity De-Lighting Tool.

Materials Authoring Guidelines : Dark Dielectric Materials
Author: Laurent Harduin, Senior Lighting Artist
Date: 2017–07–04
Checked with Unity version: 2017.1
Synopsis: This guide provides tips and calibrated albedo color measurements for authoring manufactured (non-natural) materials in the context of Unity's physically-based shading system.

Collaborate - Keep The User Safe
Author: Hollie Figueroa, Software Test Engineer for Collaborate
Date: 2017–07–19
Checked with Unity version: 2017.1
Synopsis: Collaborate is the cloud based, client-server pairing that exists between the Unity Editor and the Unity-owned servers. This guide runs you through the process of getting up and running with Collaborate, and gives advice on best practices and considerations for parallel development with Unity.

2017–10–20 Page amended with editorial review

Update to De-Lighting Tool document in Unity [2017.3]

New in Unity 2018.3 beta

Leave feedback

Search Unity User Manual: [New in Unity 2018.3]

To find User Manual pages on new and updated features in Unity 2018.3, click on the link above or search for "**NewIn20183**".

This lists User Manual pages containing Unity 2018.3's:

new features
changes to a feature's existing behavior
new additions to a feature's behavior
deprecated functionality

Release Notes

To find out more about the new features, changes, and improvements to this Unity version, see the 2018.3 Release Notes.

Upgrade Guide

If you are upgrading existing projects from an earlier version to 2018.3, read the Upgrade Guide to 2018.3 for information about how your project may be affected.

Packages Documentation

Leave feedback

A package contains features to fit the various needs of your project. This can include any core features of Unity which is installed along with the Editor, or other packages which you can install as needed. Click on any of the following currently available packages to see the documentation for that package.

Preloaded packages (installed by default along with the Editor)

Ads
Analytics
Package Manager UI
In App Purchasing
Standard Events

Other Packages

Asset Bundle Browser
Cinemachine
ProBuilder
TextMesh Pro
Xiaomi

Packages via the Package Manager was added in Unity 2018.1

Glossary

Leave feedback

2D Terms

2D Object:

A 2D **GameObject** such as a **tilemap** or **sprite**. More info

2D physics Terms

Body Type:

Defines a fixed behavior for a 2D rigid body. Can be Dynamic (the body moves under simulation and is affected by forces like gravity), Kinematic (the body moves under simulation, but and isn't affected by forces like gravity) or Static (the body doesn't move under simulation). More info

Fixed Joint 2D:

A 2D **joint** type which is completely constrained, allowing two objects to be held together. Implemented as a spring so some small motion may still occur. More info

Physics Material 2D:

Use to adjust the friction and bounce that occurs between 2D physics objects when they collide More info

Relative Joint 2D:

A 2D **joint** that allows two game objects controlled by **rigidbody** physics to maintain in a position based on each other's location. Use this **joint** to keep two objects offset from each other, at a position and angle you decide More info

AI Terms

NavMesh:

A **mesh** that Unity generates to approximate the walkable areas and obstacles in your environment for path finding and AI-controlled navigation. More info

Analytics Terms

Active Users:

Players who recently played your game. Unity **Analytics** defines an active player as someone who has played within the last 90 calendar days. More info

Ad ARP:

(Average Revenue Per User) Average Unity Ads revenue per player. More info

Ad Revenue:

Total Unity Ads revenue. More info

Ad Starts:

The number of video ads that started playing. More info

Ads per DAU:

The number of ads started per active player on a given day. More info

Age 14 and Under:

By default, Unity does not breakout **analytics** data for players under the age of 14. See **COPPA** Compliance. More info

All Spenders:

Players who have made any verified or unverified in-app purchases in their lifetime. More info

Analytics:

A data platform that provides **analytics** for your Unity game. More info

Analytics Events:

Events dispatched to the **Analytics** Service by instances of your applications. **Analytics** events contain the data that is processed and aggregated to provide insights into player behavior. More info

Application version:

Player segments based on application version or bundleid. More info

ARPDAU :

(Average Revenue Per Daily Active User) The average revenue per user who played on a given day. More info

ARPPU:

(Average Revenue Per Paying User) Average **verified IAP revenue** per user who completed a verified IAP transaction. More info

Churn:

The rate at which users are leaving your game during a specified period. Your user churn is important in estimating the lifetime value of your users. Mathematically, churn is the complement of retention (in other words: Churn + Retention = 100%). More info

Cohort:

A group of players with at least one similar characteristic. You can define and analyze different cohorts of your user base with segments. More info

Conversion Rate:

The percentage of users who complete an action or sequence of actions. More info

COPPA:

(Children's Online Privacy Protection Act) **COPPA** is a US law that applies to apps that collect personal information and are targeted to children under the age of 14. More info

Core Events:

Core events are the basic events dispatched by the Unity **Analytics** code in your game. These events, and the **analytics** based on them, become available simply by turning on Unity **Analytics** for a project. Core events include: app running, app start, and device info. More info

CTR:

(Click Through Rate) The percentage of players who click a link in an ad displayed in your game. More info

Custom Events:

Custom events are freeform events that you can dispatch when an appropriate **standard event** is not available. Custom events can have any name and up to ten parameters. Use **standard events** in preference to custom events where possible. More info

Data Explorer:

A Unity **Analytics** Dashboard page that allows you to build, view and export reports on your **Analytics** metrics and events. You can also see how metrics and custom events change over time. More info

DAU:

(Daily Active Users) The number of different players who started a session on a given day. **DAU** includes both new and returning players. More info

DAU per MAU:

(DAU/MAU) The percentage of monthly active users who play on a given day. Also known as **Sticky Factor** in the **analytics** and game industries, this metric is often used as one estimate of player engagement. More info

Day 1 Retention:

The percentage of players who returned to your game one day after playing the first time. More info

Day 30 Retention:

The percentage of players who returned to your game thirty days after playing the first time. More info

Day 7 Retention:

The percentage of players who returned to your game seven days after playing the first time. More info

Demographics:

Player segments based on reported demographics. More info

Dolphins:

Players who have spent between $5 and $19.99. More info

eCPM:

(estimated Cost Per Mille) The estimated revenue for 1000 ad impressions for your app. See What is ECPM. More info

Engagement:

Engagement is a broad measure of how players enjoy, or are otherwise invested, in your game. Impossible to measure directly, the following metrics are frequently used to estimate engagement: Retention, **DAU**, **MAU**, **DAU/MAU**, number of sessions, and session length. More info

F2P:

(Free to Play) A business model that offers users free access to a fully functional game and a significant portion of app content. Monetization strategies for these titles generally include microtransactions that allow users to access premium features and virtual goods. More info

Fill Rate:

The rate at which ads are available when you request one. More info

Funnel:

In **Analytics**, a funnel is a linear sequence of standard or custom events that you expect a player to complete in order. More info

Geography:

Player segments based on country. More info

Heatmaps:

Heatmaps are a spatial visualization of **analytics** data. More info

IAP,In App Purchase:

(In-app Purchase) Revenue from “micro-transactions” within a game. More info

Impressions:

The number of times ads are seen in your game. An impression is counted even if the ad is not completed. More info

LTV:

(Lifetime Value) The estimated value of an average player over their lifetime with your application or game. More info

MAU:

(Monthly Active Users) The number of players who started a session within the last 30 days. More info

Minnow:

A player who has spent less than $5 in their lifetime. More info

Never Monetized:

Players who have never spent real currency. More info

New Users:

Users who played your game for the first time. More info

Number of Unverified Transactions:

The total number of IAP transactions, whether or not they have been verified. More info

Number of Users:

The cumulative number of unique players over the last 90 days. Users who have not played in more than 90 days are removed from the count. More info

Number of Verified Transactions:

IAP transactions that have been verified through the appropriate app store. IAP verification is currently supported by the Apple App Store and the Google Play Store. More info

Percentage of Population:

Your player population as a percentage. Typically only useful when combined with a segment. Calculated as the percentage of the Number of Users metric who are members of a specified segment. More info

Remote Settings:

Remote settings are game variables that you can set remotely on your **Analytics** Dashboard. See Remote Settings. More info

Segment:

Segments are subsets of your player base, split apart by key differentiators. Viewing metrics and events by segment can reveal differences in-game behavior between different groups. More info

Session:

A single play or usage period. A new session is counted when a player launches your game or brings a suspended game to the foreground after 30 minutes of inactivity. More info

Sessions per User:

The average number of sessions per person playing on a given day. Also known as Average Number of Sessions per **DAU**. More info

Standard event:

Standard events are application-specific events that you dispatch in response to important player actions or milestones. **Standard events** have standardized names and defined parameter lists. More info

Sticky Factor:

An estimate of how compelling a game is to its players. A high "**sticky factor**" means that players stick with an app over time. More info

Total Daily Playing Time:

The cumulative playing time of all people playing on a given day. More info

Total Daily Playing Time per Active User:

The average playing time of people playing on a given day. More info

Total IAP Revenue:

The **total IAP revenue**, including revenue from both verified and unverified transactions. More info

Total Sessions Today:

The total number of sessions by all people playing on a given day. Also known as Total Sessions. More info

Total Verified Revenue:

Revenue from Unity Ads and verified IAP transactions. IAP verification is currently supported by the Apple App Store and the Google Play Store. More info

Unity Analytics:

See **Analytics** More info

Unity IAP:

See IAP More info

Universal Windows Platform:

An IAP feature that supports Microsoft's **In App Purchase** simulator, which allows you to test IAP purchase flows on devices before publishing your application. More info

Unknown Gender:

Players to whom you have assigned Gender.Unknown. (Players whose gender has not been reported at all are not included in this segment.) More info

Unverified IAP Revenue:

IAP revenue from sources that do not support verification and from transactions that failed verification. Transactions can fail verification because they are fraudulent or because of missing or malformed information. More info

Verified IAP Revenue:

Revenue from verified IAP transactions. IAP verification is currently supported by the Apple App Store and the Google Play Store. More info

Verified Paying Users:

Players who made verified IAP purchases. IAP verification is currently supported by the Apple App Store and the Google Play Store. More info

Whales:

Players who have spent at least $20 in their lifetime. More info

Animation Terms

1D Blend Tree:

A Blend Tree for 1D blending, which blends motion according to a single Animation Parameter. More info

2D Blend Tree:

A Blend Tree for 2D blending, which blends motion according to two **Animation Parameters**. More info

3D Object:

A 3D **GameObject** such as a cube, **terrain** or ragdoll. More info

Animation Blend Shape:

Enables you to make an object change its form by blending between two separate meshes. More info

Animation Blend Tree:

Used for continuous blending between similar **Animation Clips** based on float **Animation Parameters**. More info

Animation Clip:

Animation data that can be used for animated characters or simple animations. It is a simple "unit" piece of motion, such as (one specific instance of) "Idle", "Walk" or "Run". More info

Animation Clip Node:

A node in a Blend Tree graph that contains an **animation clip**, such as a run or walk animation. More info

animation compression:

The method of compressing animation data to significantly reduce file sizes without causing a noticable reduction in motion quality. Animation **compression** is a trade off between saving on memory and image quality. More info

Animation Curves:

Allows you to add data to an imported clip so you can animate the timings of other items based on the state of an animator. For example, for a game set in icy conditions, you could use an extra **animation curve** to control the emission rate of a **particle system** to show the player's condensing breath in the cold air. More info

Animation Event:

Allows you to add data to an imported clip which determines when certain actions should occur in time with the animation. For example, for an animated character you might want to add events to walk and run cycles to indicate when the footstep sounds should play. More info

Animation Layer:

An **Animation Layer** contains an Animation **State Machine** that controls animations of a model or part of it. An example of this is if you have a full-body layer for walking or jumping and a higher layer for upper-body motions such as throwing an object or shooting. The higher layers take precedence for the body parts they control. More info

Animation Parameters:

Used to communicate between scripting and the **Animator Controller**. Some parameters can be set in scripting and used by the controller, while other parameters are based on Custom Curves in **Animation Clips** and can be sampled using the scripting API. More info

Animation State Machine:

A graph within an **Animator Controller** that controls the interaction of Animation States. Each state references an **Animation Blend Tree** or a single **Animation Clip**. More info

Animation Transition:

Allows a **state machine** to switch or blend from one animation state to another. Transitions define how long a blend between states should take, and the conditions that activate them. More info

Animator Component:

A **component** on a model that animates that model using the Animation system. The **component** has a reference to an **Animator Controller asset** that controls the animation. More info

Animator Controller:

Controls animation through **Animation Layers** with **Animation State Machines** and **Animation Blend Trees**, controlled by **Animation Parameters**. The same **Animator Controller** can be referenced by multiple models with **Animator components**. More info

Animator Override Controller:

Allows you to create multiple variants of an **Animator Controller**, with each variant using a different set of animations, while retaining the original Controller's structure, parameters and logic. More info

Animator Window:

The window where the **Animator Controller** is visualized and edited. More info

Avatar:

An interface for **retargeting** animation from one rig to another. More info

Avatar Mask:

A specification for which body parts to include or exclude for an animation rig. Used in **Animation Layers** and in the importer. More info

Bind-pose:

The pose at which the character was modelled.

blend:

Transition from one animation to another animation smoothly and seamlessly, such as blending a character's walking and running animations according to the character's speed.

Blend Node:

A node in a Blend Tree graph that blends **animation clip** nodes. More info

Body Transform:

The mass center of the character. It is used in for animation **retargeting** and provides the most stable displacement model. More info

Direct Blend Tree:

A Blend Tree that allows you to map animator parameters directly to the weight of a Blend Tree child. This is useful when you want to have exact control over the animations that are being blended rather than blend them indirectly using one or two parameters (in the case of 1D and 2D blend trees). More info

forward kinematics:

A method of posing a skeleton for animation by rotating the **joint** angles to predetermined values. The position of a child **joint** changes according to the rotation of its parent and so the end point of a chain of **joints** can be determined from the angles and relative positions of the individual **joints** it contains.

Human template:

A pre-defined bone-mapping. Used for matching bones from FBX files to the **Avatar**. More info

Humanoid animation:

An animation using humanoid skeletons. Humanoid models generally have the same basic structure, representing the major articulate parts of the body, head and limbs. This makes it easy to map animations from one humanoid skeleton to another, allowing **retargeting** and inverse **kinematics**. More info

inverse kinematics (IK):

The automatic calculation of **joint** angles (eg. the shoulder and elbow joint of an arm) so that the end point (eg. the hand) reaches a desired point in space. In contrast to **Forward Kinematics** More info

keyframe:

A frame that marks the start or end point of a transition in an animation. Frames in between the **keyframes** are called inbetweens.

Keyframe Reduction:

A process that removes redundant **keyframes**. More info

kinematics:

The geometry that describes the position and orientation of a character's **joints** and bodies. Used by inverse **kinematics** to control character movement.

Loop Pose:

An **animation clip** setting that blends the end and start of an animation to create a seamless join. More info

Muscle definition:

This allows you to have more intuitive control over the character's skeleton. When an **Avatar** is in place, the Animation system works in muscle space, which is more intuitive than bone space. More info

Ping Pong:

To repeatedly play an animation to the end, then in reverse back to the beginning, in a loop.

Playable Graph:

An API for controlling **Playables**. **Playable Graphs** allow you to create, connect and destroy **playables**. More info

Playables:

An API that provides a way to create tools, effects or other gameplay mechanisms by organizing and evaluating data sources in a tree-like structure known as the PlayableGraph. More info

Retargeting:

Applying animations created for one model to another. More info

Rigging:

The process of building a skeleton hierarchy of **joints** for your **mesh**. Performed with an external tool, such as Blender or Autodesk Maya. More info

Root Motion:

Motion of character's **root node**, whether it's controlled by the animation itself or externally. More info

Root node:

A transform in an animation hierarchy that allows Unity to establish consistency between **Animation clips** for a generic model. It also enables Unity to properly blend between Animations that have not been authored "in place" (that is, where the whole Model moves its world position while animating). More info

Root Transform:

A projection on the Y plane of the Body Transform, computed at runtime. At every frame, a change in the **Root Transform** is computed. This change in transform is then applied to the Game Object to make it move. More info

Scene:

A **Scene** contains the environments and menus of your game. Think of each unique **Scene** file as a unique level. In each **Scene**, you place your environments, obstacles, and decorations, essentially designing and building your game in pieces. More info

Skinning:

The process of binding bone **joints** to the vertices of a character's **mesh** or 'skin'. Performed with an external tool, such as Blender or Autodesk Maya. More info

State Machine:

The set of states in an **Animator Controller** that a character or animated **GameObject** can be in, along with a set of transitions between those states and a variable to remember the current state. The states available will depend on the type of gameplay, but typical states include things like idling, walking, running and jumping. More info

State Machine Behaviour:

A script that attaches to a state within a **state machine** to control what happens when the **state machine** enters, exits or remains within a state, such as play sounds as states are entered. More info

T-pose:

The pose in which the character has their arms straight out to the sides, forming a "T". The required pose for the character to be in, in order to make an **Avatar**.

Target matching:

A scripting function that allows you to move characters in such a way that a hand or foot lands in a certain place at a certain time. For example, the character may need to jump across stepping stones or jump and grab an overhead beam. More info

Transition:

The blend from one state to another in a **state machine**, such as transitioning a character from a walk to a jog animation. Transitions define how long the blend between states should take, and the conditions that activate the blend. More info

Translate DoF:

The three degrees-of-freedom associated with translation (movement in X,Y & Z) as opposed to rotation.

Assets Terms

Asset:

Any media or data that can be used in your game or project. An **asset** may come from a file created outside of Unity, such as a 3D model, an audio file or an image. You can also create some **asset** types in Unity, such as an **Animator Controller**, an Audio Mixer or a **Render Texture**. More info

Asset Package:

A collection of files and data from Unity projects, or elements of projects, which are compressed and stored in one file, similar to Zip files. Packages are a handy way of sharing and re-using Unity projects and collections of **assets**. More info

Asset Server:

Legacy - An **asset** and **version control** system with a graphical user interface integrated into Unity. Enables team members to work together on a project on different computers. More info

Asset Store:

A growing library of free and commercial **assets** created by Unity and members of the community. Offers a wide variety of **assets**, from textures, models and animations to whole project examples, tutorials and Editor extensions. More info

Billboard Asset:

An **asset** that is a collection of pre-rendered images of a more complicated **Mesh** intended for use with the **Billboard** Renderer, in order to render an object at some distance from a **Camera** at a lower level of detail (LOD) to save on **rendering** time. More info

Billboard Renderer:

Renders **Billboard Assets**, either from a pre-made **Asset** (exported from SpeedTree) or from a custom-created file that you create using a script at runtime or from a custom editor, for example. More info

Cache Server:

A standalone app that you can run on your local computer that stores the imported **Asset** data to reduce the time it takes to import **Assets**. More info

Model:

A 3D model representation of an object, such as a character, a building, or a piece of furniture. More info

Model file:

A file containing a 3D data, which may include definitions for meshes, bones, animation, materials and textures. More info

Prefab:

An **asset** type that allows you to store a **GameObject** complete with **components** and properties. The **prefab** acts as a template from which you can create new object instances in the **scene**. More info

Standard Asset:

A collection of useful **assets** supplied with Unity. Unity ships with multiple **Standard Asset** such as 2D, **Cameras**, Characters, CrossPlatformInput, Effects, Environment, ParticleSystems, Prototyping, Utility, and Vehicles. More info

Audio Terms

Audio Clip:

A container for audio data in Unity. Unity supports mono, stereo and multichannel audio **assets** (up to eight channels). Unity can import .aif, .wav, .mp3, and .ogg audio file format, and .xm, .mod, .it, and .s3m tracker module formats. More info

Audio Distortion Filter:

An **audio filter** that distorts the sound from an **Audio Source** or sounds reaching the **Audio Listener** by simulating the sound of a low quality radio transmission. More info

Audio Effect:

Any effect that can modify the output of Audio Mixer **components**, such as filtering frequency ranges of a sound or applying reverb. More info

Audio Filter:

Any **audio filter** that distorts the sound from an **Audio Source** or sounds reaching the **Audio Listener**. More info

Audio High Pass Filter:

An **audio filter** that passes high frequencies of an AudioSource and cuts off signals with frequencies lower than the Cutoff Frequency. More info

Audio Listener:

A **component** that acts like a microphone, receiving sound from **Audio Sources** in the **scene** and outputting to the computer speakers. More info

Audio Low Pass Filter:

An **audio filter** that passes low frequencies of an **Audio Source** or all sound reaching an **Audio Listener** while removing frequencies higher than the Cutoff Frequency. More info

Audio Source:

A **component** which plays back an **Audio Clip** in the **scene** to an **audio listener** or through an audio mixer. More info

Audio Spatializer:

A **plugin** that changes the way audio is transmitted from an **audio source** into the surrounding space. It takes the source and regulates the gains of the left and right ear contributions based on the distance and angle between the AudioListener and the AudioSource. More info

Distortion Effect:

An **audio effect** that modifies the sound by squashing and clipping the waveform to produce a rough, harsh result. More info

Doppler Factor:

An audio setting that allows you to control how much the velocity of an object (relative to the audio listener) affects the pitch of any **audio sources** attached to it. More info

Dry Level:

An audio setting that allows you to set the mix level of unprocessed signal in output in mB.

Dry Mix:

An audio setting that allows you to set the volume of the original signal to pass to output.

Play On Awake:

Set this to true to make an **Audio Source** start playing on awake More info

Core Terms

build:

The process of compiling your project into a format that is ready to run on a specific platform or platforms. More info

Managed plugin:

A managed .NET assembly that is created with tools like Visual Studio for use in Unity. More info

Native plugin:

A platform-specific native code library that is created outside of Unity for use in Unity. Allows you can access features like OS calls and third-party code libraries that would otherwise not be available to Unity. More info

Perforce:

A **version control** system for file change management. More info

Development Build:

A **development build** includes debug symbols and enables the **Profiler**. More info

Editor Terms

Anchor:

A **UI** layout tool that fixes a **UI** element to a parent element. Anchors are shown as four small triangular handles in the **Scene View** and anchor information is also shown in the **Inspector**. More info

Console window:

A Unity Editor window that shows errors, warnings and other messages generated by Unity, or your own **scripts**. More info

Input Manager:

A Unity window where you can define all the different input axes, buttons and controls for your project. More info

Inspector:

A Unity window that displays information about the currently selected **GameObject**, **Asset** or **Project Settings**, alowing you to inspect and edit the values. More info

Player Settings:

A settings manager that lets you set various player-specific options for the final game built by Unity. More info

Project View:

A view that shows the contents of your **Assets** folder (Project tab) More info

Property Drawer:

A Unity feature that allows you to customize the look of certain controls in the **Inspector** window by using attributes on your **scripts**, or by controlling how a specific Serializable class should look More info

Scene View:

An interactive view into the world you are creating. You use the **Scene View** to select and position scenery, characters, **cameras**, lights, and all other types of Game Object. More info

Time Manager:

A Unity Settings Manager that lets you set a number of properties that control timing within your game. More info

zoom:

A **camera** control that lets you scale the view on your screen. To zoom a **camera** in the Unity Editor, press Alt + right click and drag. More info

General Terms

Billboard:

A textured **2D object** that rotates as it, or the **Camera**, moves so that it always faces the **Camera**. See Billboard Renderer

Billboarding:

The process of constantly altering the **2D object**'s rotation so that it always faces the **Camera**. **Billboarding** is often used as a level-of-detail method for objects far away from the **Camera**. See Billboard Renderer

compression:

A method of storing data that reduces the amount of storage space it requires. See Texture Compression, Animation Compression, Audio Compression, Build Compression.

Extrapolate, Extrapolation:

The process of storing the last few known values and using them to predict future values. Used in animation, physics and multiplayer.

first person shooter:

A common game genre, featuring a first-person view of a 3D world, and gun-based combat with other players or NPCs.

FPS:

See **first person shooter**, **frames per second**.

game console:

A device that runs and displays video games.

game controller:

A device to control objects and characters in a game.

Velocity:

A vector that defines the speed and direction of motion of a **Rigidbody**

Version Control:

A system for managing file changes. You can use Unity in conjunction with most common **version control** tools, including **Perforce**, Git, Mercurial and PlasticSCM. More info

Viewport:

The user's visible area of an app on their screen.

Profiler:

A window that helps you to optimize your game. It shows how much time is spent in the various areas of your game. For example, it can report the percentage of time spent **rendering**, animating or in your game logic. More info

Project Settings:

A broad collection of settings which allow you to configure how Physics, Audio, **Networking**, Graphics, Input and many other areas of your project behave. More info

console:

See **game console**

Deferred shading:

A **rendering path** that places no limit on the number of lights that can affect a **GameObject**. All lights are evaluated per-pixel, which means that they all interact correctly with normal maps and so on. Additionally, all lights can have cookies and shadows. More info

FBX:

See **Model File** More info

GameObject:

The fundamental object in Unity **scenes**, which can represent characters, props, scenery, **cameras**, waypoints, and more. A **GameObject**'s functionality is defined by the **Components** attached to it. More info

Joy Num:

An **Input Manager** property that defines which joystick will be used. More info

Animation Key:

The value of an **animatable property**, set at a specific point in time. Setting at least two keys for the same property creates an animation. More info

Input Key:

A key on a keyboard relating to the Input class. More info

Mask:

Can refer to a **Sprite Mask**, a **UI Mask**, or a **Layer Mask** More info

Layer Mask:

A value defining which layers to include or exclude from an operation, such as **rendering**, **collision** or your own code. More info

Sprite Mask:

A texture which defines which areas of an underlying image to reveal or hide. More info

Object:

See **GameObject**.

Parent:

An object that contains child objects in a hierarchy. When a **GameObject** is a Parent of another **GameObject**, the Child **GameObject** will move, rotate, and scale exactly as its Parent does. You can think of parenting as being like the relationship between your arms and your body; whenever your body moves, your arms also move along with it. More info

Plugin:

A set of code created outside of Unity that creates functionality in Unity. There are two kinds of **plugins** you can use in Unity: **Managed plugins** (managed .NET assemblies created with tools like Visual Studio) and Native **plugins** (platform-specific native code libraries). More info

Transform Component:

A Transform **component** determines the Position, Rotation, and Scale of each object in the **scene**. Every **GameObject** has a Transform. More info

Tree:

A **GameObject** and associated **Component** that allows you to add tree **assets** to your **scene**. You can add branch levels and leaves to trees in the Tree **Inspector** window. More info

World:

The area in your **scene** in which all objects reside. Often used to specify that coordinates are world-relative, as opposed to object-relative.

General Terms

Interpolate, Interpolation:

The process of calculating values in-between two defined values. Used in animation (between keyframes), physics (between physics time-steps), and multiplayer (between network updates)

Graphics Terms

Ambient GI:

A ambient light source generated by the Global Illumination (GI) system that provides omni-directional light to all objects in the **scene** equally. More info

Ambient occlusion:

A method to approximate how much ambient lighting (lighting not coming from a specific direction) can hit a point on a surface. More info

Aniso Level:

The anisotropic filtering (AF) level of a texture. Allows you to increase texture quality when viewing a texture at a steep angle. Good for floor and ground textures. More info

Antialiasing:

A technique for decreasing artifacts, like jagged lines (jaggies), in images to make them appear smoother. More info

Aspect Ratio:

The relationship of an image's proportional dimensions, such as its width and height.

ASTC:

Adaptive Scalable Texture **Compression** (ASTC) A block-based **texture format** that compresses textures to significantly reduce file sizes without cau sing a noticable reduction in image quality. More info

ATC:

AMD's texture **compression** format for handheld devices to save on power, memory and bandwidth

Baked Lights:

A **Light Mode** for creating local ambience, rather than fully featured lights for increasing brightness in dark areas without needing to adjust all of the lighting within a **Scene**. Unity pre-calculates the illumination from these lights before run time, and does not include them in any run-time lighting calculations. More info

Bloom:

A post-processing effect used to reproduce an imaging artifact of real-world **cameras**. The effect produces fringes of light extending from the borders of bright areas in an image, contributing to the illusion of an extremely bright light overwhelming the **camera** or eye capturing the **scene**.

Bounds:

The coordinates that define the **bounding volume**. Bounds are pre-calculated on import from the **Mesh** and animations in the **model file**, and are displayed as a wireframe around the model in the **Scene View**.

Branch Group:

A tree node that generates branches and fronds. Its properties appear when you have selected a branch or leaf node. More info

Bump map:

An image texture used to represent geometric detail across the surface of a **mesh**, for example bumps and grooves. Can be represented as a height map or a normal map. More info

Camera:

A **component** which creates an image of a particular viewpoint in your **scene**. The output is either drawn to the screen or captured as a texture. More info

CGPROGRAM:

A block of **shader** code for controlling **shaders** using NVIDIA's Cg (C for graphics) programming language. More info

clipping plane:

A plane that limits how far or close a **camera** can see from its current position. A **camera**'s viewable range is between the far and near **clipping planes**. See far **clipping plane** and near **clipping plane**. More info

Color Animation:

Any color property of a **component** can be animated. Also **particle system** colors can be animated using the "Color over lifetime" module.

component:

A functional part of a **GameObject**. A **GameObject** can contain any number of **components**. Unity has many built-in **components**, and you can create your own by writing **scripts** that inherit from MonoBehaviour. More info

Content:

The objects being rendered in your **scene**. More info

Context:

The lighting that exists in the **scene** which affects the object being lit. More info

Cubemap:

A collection of six square textures that can represent the reflections in an environment or the **skybox** drawn behind your geometry. The six squares form the faces of an imaginary cube that surrounds an object; each face represents the view along the directions of the world axes (up, down, left, right, forward and back). More info

Culling Mask:

Allows you to includes or omit objects to be rendered by a **Camera**, by Layer.

Depth of Field:

A post-processing effect that simulates the focus properties of a **camera** lens. More info

Diffuse shader:

A old type of **shader** used in earlier versions of Unity. Replaced by the **Standard Shader** from Unity 5 onwards. More info

Distance Shadowmask:

A version of the **Shadowmask** lighting mode that includes high quality shadows cast from static **GameObjects** onto dynamic **GameObjects**. More info

Dynamic Batching:

An automatic Unity process which attempts to render multiple meshes as if they were a single **mesh** for optimized graphics performance. The technique transforms all of the **GameObject** vertices on the CPU and groups many similar vertices together. More info

Dynamic receiver:

A dynamic **GameObject** that is receiving a shadow from another static or dynamic **GameObject** More info

Enlighten:

The lighting system by Geomerics used in Unity for computing global illumination (GI). More info

ETC:

(Ericsson Texture Compression) A block-based **texture format** that compresses textures to significantly reduce file sizes without causing a noticable reduction in image quality. More info

Exposure value:

A value that represents a combination of a **camera**'s shutter speed and f-number. It is essentially a measurement of exposure such that all combinations of shutter speed and f-number that yield the same level of exposure have the same EV.

Extrude Edges:

A Texture property that enables you to define how much area to leave around a **sprite** in a generated **mesh**.

Far clipping plane:

The maximum draw distance for a **camera**. Geometry beyond the plane defined by this value is not rendered. The plane is perpendicular to the **camera**'s forward (Z) direction.

Flare:

The source **asset** used by **Lens Flare Components**. The Flare itself is a combination of a texture file and specific information that determines how the Flare behaves. More info

Flare Fade Speed:

A Lighting window property that allows you to set the time (in seconds) over which **lens flares** fade from view after initially appearing. More info

Flare Layer:

A **component** that you can attach to **Cameras** to make **Lens Flares** appear in the image. By default, a **Flare Layer** is already attached to a **Camera**.

Flare Texture:

A texture containing images used by a Flare **asset**. It must be arranged according to one of the TextureLayout options.

Fog:

A post-processing effect that overlays a color onto objects depending on the distance from the **camera**. Use this to simulate fog or mist in outdoor environments, or to hide clipping of objects near the **camera**'s far clip plane. More info

Forward Rendering:

A **rendering path** that renders each object in one or more passes, depending on lights that affect the object. Lights themselves are also treated differently by **Forward Rendering**, depending on their settings and intensity. More info

fragment shader:

The "per-pixel" part of **shader** code, performed every **pixel** that an object occupies on-screen. The **fragment shader** part is usually used to calculate and output the color of each **pixel**. More info

frame:

A single image from a sequence of images that represent moving graphics. While you game is running, the **camera** in your game renders frames to the screen as fast as possible. May also refer to a frame from a video clip, or **sprite** animation frames. See: **frames per second** (FPS).

frames per second:

The frequency at which consecutive frames are displayed in a running game. More info

Fresnel Effect:

An effect representing the increase in reflectivity on objects when light hits at grazing angles. In Unity, the **Standard Shader** handles this indirectly, depending on a material's smoothness. Smooth surfaces have a stronger Fresnel, totally rough surfaces have no Fresnel. More info

Frustum:

The region of 3D space that a perspective **camera** can see and render. In the **Scene view**, the frustum is represented by a truncated rectangular pyramid with its top at the **camera__'s near** clipping plane__A plane that limits how far or close a camera can see from its current position. A camera's viewable range is between the far and near clipping planes. See far clipping plane and near clipping plane. More info
See in Glossary and its base at the **camera__'s far** clipping plane__. More info

GI Cache:

The cached intermediate files used when Unity precomputes real-time GI, and when baking Static **Lightmaps**, **Light Probes** and **Reflection Probes**. Unity keeps this cache to speed up computation. More info

Gizmo:

A graphic overlay associated with a **GameObject** in a **Scene**, and displayed in the **Scene View**. Built-in **scene** tools such as the move tool are **Gizmos**, and you can create custom **Gizmos** using textures or scripting. Some **Gizmos** are only drawn when the **GameObject** is selected, while other **Gizmos** are drawn by the Editor regardless of which **GameObjects** are selected. More info

Gravity Modifier:

A **Particle System** property that scales the gravity value set in the physics manager. A value of zero switches gravity off. More info

Group Seed:

A tree property that varies the procedural generation of branch and leaf groups. More info

Growth Angle:

A tree property that sets the initial angle of branch or leaf group growth relative to the parent. More info

Growth Scale:

A tree property that sets the scale of branch and leaf group nodes along the parent node. More info

Halo:

The glowing light areas around light sources, used to give the impression of small dust particles in the air. More info

Hard Shadows:

A shadow property that produces shadows with a sharp edge. Hard shadows are not particularly realistic compared to Soft Shadows but they involve less processing, and are acceptable for many purposes. More info

HDR:

high dymanic range

HDRI:

high dynamic range image

Irradiance Budget:

A **lightmap** property that determines the precision of the incoming light data used to light each texel in the **lightmap** More info

Irradiance Quality:

A **lightmap** property that sets the number of rays that are cast and used to compute which clusters affect a given output **lightmap** texel. More info

Layer:

Layers in Unity can be used to selectively opt groups of **GameObjects** in or out of certain processes or calculations. This includes **camera rendering**, lighting, physics **collisions**, or custom calculations in your own code. More info

Lens Dirt:

A Bloom effect property that applies a fullscreen layer of smudges or dust to diffract the Bloom effect. This is commonly used in modern **first person shooters**. More info

Lens Flare:

A **component** that simulates the effect of lights refracting inside a **camera** lens. Use a **Lens Flare** to represent very bright lights or add atmosphere to your **scene**. More info

Light Mode:

A Light property that defines the use of the Light. Can be set to Realtime, Baked and Mixed. More info

Light Probe:

Light probes store information about how light passes through space in your **scene**. A collection of **light probes** arranged within a given space can improve lighting on moving objects and static **LOD** scenery within that space. More info

Light Probe Group:

A **component** that enables you to add **Light Probes** to **GameObjects** in your **scene**. More info

Light Probe Proxy Volume:

A **component** that allows you to use more lighting information for large dynamic **GameObjects** that cannot use baked **lightmaps** (for example, large Particle Systems or skinned Meshes). More info

Lightmap:

A pre-rendered texture that contains the effects of light sources on static objects in the **scene**. **Lightmaps** are overlaid on top of **scene** geometry to create the effect of lighting. More info

Lightmapper:

A tool in Unity that bakes **lightmaps** according to the arrangement of lights and geometry in your **scene**. More info

Line Renderer:

A **component** that takes an array of two or more points in 3D space and draws a straight line between each one. You can use a single **Line Renderer component** to draw anything from a simple straight line to a complex spiral. More info

LOD:

See level of detail More info

LOD Group:

A **component** to manage level of detail (LOD) for **GameObjects**. More info

material:

An **asset** that defines how a surface should be rendered, by including references to the Textures it uses, tiling information, Color tints and more. The available options for a Material depend on which **Shader** the Material is using. More info

Mesh:

The main graphics primitive of Unity. Meshes make up a large part of your 3D worlds. Unity supports triangulated or Quadrangulated polygon meshes. Nurbs, Nurms, Subdiv surfaces must be converted to polygons. More info

Mesh Filter:

A **mesh component** that takes a **mesh** from your **assets** and passes it to the **Mesh** Renderer for **rendering** on the screen. More info

Mesh Renderer:

A **mesh component** that takes the geometry from the **Mesh Filter** and renders it at the position defined by the object's Transform **component**. More info

Motion Blur:

A common post-processing effect that simulates the blurring of an image when objects filmed by a **camera** are moving faster than the **camera**'s exposure time. More info

Near clipping plane:

A plane that limits how close a **camera** can see from its current position. The plane is perpendicular to the **camera**'s forward (Z) direction. More info

Normal:

The direction perpendicular to the surface of a **mesh**, represented by a Vector. Unity uses normals to determine object orientation and apply shading. More info

Normal map : A type of **Bump Map** texture that allows you to add surface detail such as bumps, grooves, and scratches to a model which catch the light as if they are represented by real geometry. More info

Occlusion Area:

A **component** which defines a 3D space within which to apply **Occlusion Culling** to moving objects. More info

Occlusion Culling:

A Unity feature that disables **rendering** of objects when they are not currently seen by the **camera** because they are obscured (occluded) by other objects. More info

Occlusion Portal:

An occlusion primitive that acts as an occluder, but can be switched on or off. Commonly used to represent opaque doors in games. More info

OpenGL Core:

The back-end Unity uses to support the latest OpenGL features on Windows, MacOS X and Linux. More info

Orthographic 3D:

A common type of **camera** view used in games that give you a bird's-eye view of the action, and is sometimes called "2.5D". More info

particle:

A small, simple image or **mesh** that is emitted by a **particle system**. A **particle system** can display and move particles in great numbers to represent a fluid or amorphous entity. The effect of all the particles together creates the impression of the complete entity, such as smoke. More info

particle system:

A **component** that simulates fluid entities such as liquids, clouds and flames by generating and animating large numbers of small 2D images in the **scene**. More info

Physically Based Shading:

An advanced lighting model that simulates the interactions between materials and light in a way that mimics reality. More info

pixel:

The smallest unit in a computer image. **Pixel** size depends on your screen resolution. **Pixel** lighting is calculated at every screen **pixel**. More info

Player Log:

The .log file created by a Standalone Player that contains a record of events, such as script execution times, the compiler version, and AssetImport time. Log files can help diagnose problems. More info

PVRTC:

PowerVR Texture **Compression** (PVRTC) is a fixed-rate **texture format** that compresses textures to significantly reduce file sizes without causing a noticable reduction in image quality. More info

Quad:

A primitive object that resembles a plane but its edges are only one unit long, it uses only 4 vertices, and the surface is oriented in the XY plane of the local coordinate space. More info

Quaternion:

Unity's standard way of representing rotations as data. When writing code that deals with rotations, you should usually use the **Quaternion** class and its methods. More info

Real-time light:

A Lighting Mode for Lights that need to change their properties or which are spawned via **scripts** during gameplay. Unity calculates and updates the lighting of these Lights every frame at run time. They can change in response to actions taken by the player, or events which take place in the **Scene**. More info

Reflection Probe:

A **rendering component** that captures a spherical view of its surroundings in all directions, rather like a **camera**. The captured image is then stored as a **Cubemap** that can be used by objects with reflective materials. More info

Render Texture:

A special type of Texture that is created and updated at runtime. To use them, first create a new **Render Texture** and designate one of your **Cameras** to render into it. Then you can use the **Render Texture** in a Material just like a regular Texture. More info

Rendering:

The process of drawing graphics to the screen (or to a render texture). By default, the main **camera** in Unity renders its view to the screen. More info

Rendering Mode:

A **Standard Shader** Material parameter that allows you to choose whether the object uses transparency, and if so, which type of blending mode to use. More info

Rendering Path:

The technique Unity uses to render graphics. Choosing a different path affects the performance of your game, and how lighting and shading are calculated. Some paths are more suited to different platforms and hardware than others. More info

Ribbon:

A **particle system** property that connects particles together to create a ribbon effect. More info

shader:

A small script that contains the mathematical calculations and algorithms for calculating the Color of each **pixel** rendered, based on the lighting input and the Material configuration. More info

ShaderLab:

Unity's declarative language for writing **shaders**. More info

Shadowmask:

A Texture that shares the same UV layout and resolution with its corresponding **lightmap**. More info

Skybox:

A special type of Material used to represent skies. Usually six-sided. More info

Soft Shadows:

A shadow property that produces shadows with a soft edge. Soft shadows are more realistic compared to Hard Shadows and tend to reduce the "blocky" aliasing effect from the shadow map, but they require more processing. More info

Sorting Fudge:

A **Particle System** property that sets the bias of the **particle system** sort ordering. Lower values increase the relative chance that **particle systems** are drawn over other transparent **GameObjects**, including other **particle systems**. More info

Spatial Mapping:

The process of mapping real-world surfaces into the virtual world. More info

Sprite:

A 2D graphic objects. If you are used to working in 3D, **Sprites** are essentially just standard textures but there are special techniques for combining and managing **sprite** textures for efficiency and convenience during development. More info

Sprite Packer:

A facility that packs graphics from several **sprite** textures tightly together within a single texture known as an atlas. Unity provides a **Sprite Packer** utility to automate the process of generating atlases from the individual **sprite** textures. More info

Sprite Renderer:

A **component** that lets you display images as **Sprites** for use in both 2D and 3D **scenes**. More info

Standard Shader:

A built-in **shader** for **rendering** real-world objects such as stone, wood, glass, plastic and metal. Supports a wide range of **shader** types and combinations. More info

Static Batching:

A technique Unity uses to draw **GameObjects** on the screen that combines static (non-moving) **GameObjects** into big Meshes, and renders them in a faster way. More info

Static receiver:

A static **GameObject** that is receiving a shadow from another static or dynamic **GameObject** More info

subshader:

Each **shader** in Unity consists of a list of **subshaders**. When Unity has to display a **mesh**, it will find the **shader** to use, and pick the first **subshader** that runs on the user's graphics card. More info

surface shader:

Unity's code generation approach that makes it much easier to write lit **shaders** than using low level vertex/**pixel** **shader** programs. More info

Terrain:

The landscape in your **scene**. A **Terrain GameObject** adds a large flat plane to your **scene** and you can use the **Terrain**'s **Inspector** window to create a detailed landscape. More info

Text Mesh:

A **Mesh component** that displays a Text string More info

texture:

An image used when **rendering** a **GameObject**, **Sprite**, or **UI** element. Textures are often applied to the surface of a **mesh** to give it visual detail. More info

Texture Compression:

3D Graphics hardware requires Textures to be compressed in specialised formats which are optimised for fast Texture sampling. More info

Texture Import Inspector:

An **Inspector** that allows you to define how your images are imported from your Project's **Assets** folder into the Unity Editor. More info

Texture Overrides:

Platform-specific settings that allow you to set the resolution, file size with associated memory size requirements, **pixel** dimensions, and quality of your Textures for each target platform. More info

Tile:

A simple class that allows a **sprite** to be rendered on a **Tilemap**. More info

Tilemap:

A **GameObject** that allows you to quickly create 2D levels using tiles and a grid overlay. More info

Tonemapping:

The process of remapping **HDR** values of an image into a range suitable to be displayed on screen. More info

Trail Renderer:

A visual effect that lets you to make trails behind **GameObjects** in the **Scene** as they move. More info

vertex shader:

A program that runs on each vertex of a 3D model when the model is being rendered. More info

VideoCapture:

A Unity API that allows you to record videos directly to the file system in the MP4 format. More info

Voxel:

A 3D **pixel**. More info

Web Camera:

A Unity asynchronous API that provides the ability to take pictures and record videos. More info

WebGL:

A JavaScript API that renders 2D and 3D graphics in a web browser. The Unity **WebGL** build option allows Unity to publish content as JavaScript programs which use HTML5 technologies and the **WebGL rendering** API to run Unity content in a web browser. More info

wind zone:

A **GameObject** that adds the effect of wind to your **terrain**. For instance, Trees within a **wind zone** will bend in a realistic animated fashion and the wind itself will move in pulses to create natural patterns of movement among the tree. More info

level of detail, LOD:

A **rendering** optimisation technique that allows you to reduce the number of triangles rendered for an object in stages as its distance from the **camera** increases. As long as your objects aren't all close to the **camera** at the same time, **LOD** will reduce the load on the hardware and improve **rendering** performance. More info

Lighting Terms

Mixed Lighting:

A **Light Mode** for creating indirect lighting, **shadowmasks** and subtractive lighting. Indirect lighting gets baked into **lightmaps** and **light probes**. **Shadowmasks** and **light probes** occlusion get generated for baked shadows. More info

Multiplayer Terms

High Level API, LOD:

A system for building multiplayer capabilities for Unity games. It is built on top of the lower level transport real-time communication layer, and handles many of the common tasks that are required for multiplayer games. More info

Host:

In a multiplayer network game without a dedicated server, one of the peers in the game acts as the center of authority for the game. This peer is called the "host". It runs a server and a "local client", while the other peers each run a "remote client". More info

Player Object:

A High Level API (HPAPI) object that represents the player on the server and has the ability to run commands (which are secure client-to-server remote procedure calls) from the player's client. More info

netId:

A unique identifier given to an object instance to track it between networked clients and the server. More info

Network Manager:

A **Networking component** that manages the network state of a Project. More info

NetworkIdentity:

A **Networking component** that allows you to assign an identity to your **GameObject** for the network to recognise it as a Local Player **GameObject** or a Server Only **GameObject**. More info

Networking:

The Unity system that enables multiplayer gaming across a computer network. More info

NetworkManagerHUD:

A **Networking component** that creates a **UI** menu that allows you to control the network state of your game using your **Network Manager**. More info

NetworkTransform:

A **Networking component** that allows you to synchronise the movements of **GameObjects** across the network. More info

Physics Terms

Bounding volume:

A closed shape representing the edges and faces of a **collider** or trigger. More info

box collider:

A cube-shaped **collider component** that handles **collisions** for **GameObjects** like dice and ice cubes. More info

capsule collider:

A capsule-shaped **collider component** that handles **collisions** for **GameObjects** like barrels and character limbs. More info

Center of Mass:

Represents the average position of all mass in a **rigidbody** for the purposes of physics calculations. By default it is computed from all **colliders** belonging to the **rigidbody**, but can be modified via script. More info

Character Controller:

A simple, capsule-shaped **collider component** with specialized features for behaving as a character in a game. Unlike true **collider components**, a **rigidbody** is not needed and the momentum effects are not realistic. More info

Character Joint:

An extended ball-socket **joint** which allows a **joint** to be limited on each axis. Mainly used for Ragdoll effects. More info

Cloth:

A **component** that works with the Skinned **Mesh** Renderer to provide a physics-based solution for simulating fabrics. More info

Collider:

An invisible shape that is used to handle physical **collisions** for an object. A **collider** doesn't need to be exactly the same shape as the object's **mesh** - a rough approximation is often more efficient and indistinguishable in gameplay. More info

Collision:

A **collision** occurs when the **physics engine** detects that the **colliders** of two **GameObjects** make contact or overlap, when at least one has a **rigidbody component** and is in motion. More info

Collision Detection:

An automatic process performed by Unity which determines whether a moving **GameObject** with a **rigidbody** and **collider component** has come into contact with any other **colliders**. More info

Configurable Joint:

An extremely customisable **joint** that other **joint** types are derived from. It can be used to create anything from adapted versions of existing **joints** to custom designed and highly specialised **joints**. More info

Constant Force:

A simple **component** for adding a **constant force** or torque to game objects with a **Rigidbody**. More info

Constraints:

Settings on **Joint components** which limit movement or rotation. The type and number of constraints vary depending on the type of **Joint**. More info

Contact Distance:

A **joint** limit property that sets the minimum distance tolerance between the **joint** position and the limit at which the limit will be enforced. More info

Damping Ratio:

A **joint** setting to control spring oscillation. A higher **damping ratio** means the spring will come to rest faster. More info

Dynamic Friction:

A **Physic Material** property that defines the friction for a **Rigidbody** when it's in motion. Lower values mean less friction, so a setting of zero represents slipping on ice. More info

Fixed Joint:

A **joint** type that is completely constrained, allowing two objects to be held together. Implemented as a spring so some motion may still occur. More info

Fixed Timestep:

A customizable framerate-independent interval that dictates when physics calculations and FixedUpdate() events are performed. More info

High Twist Limit:

The higher limit of a **Character Joint**. More info

Hinge Joint:

A **joint** that groups together two **Rigidbodies**, constraining them to move like they are connected by a hinge. It is perfect for doors, but can also be used to model chains, pendulums and so on. More info

joint:

A physics **component** allowing a dynamic connection between **rigidbodies**, usually allowing some degree of movement such as a hinge. More info

Low Twist Limit:

A **joint** property that sets the lower limit of a **joint**. More info

Mesh Collider:

A free-form **collider component** which accepts a **mesh** reference to define its **collision** surface shape. More info

Physic Material:

A physics **asset** for adjusting the friction and bouncing effects of colliding objects. More info

Physics Engine:

A system that simulates aspects of physical systems so that objects can accelerate correctly and be affected by **collisions**, gravity and other forces. More info

Rig:

A skeletal hierarchy of **joints** for a **mesh**. More info

Rigidbody:

A **component** that allows a **GameObject** to be affected by simulated gravity and other forces. More info

Self Collision:

A cloth property that prevents cloth from penetrating itself. More info

Sphere Collider:

A sphere-shaped **collider component** that handles **collisions** for **GameObjects** like balls or other things that can be roughly approximated as a sphere for the purposes of physics. More info

Spring Joint:

A **joint** type that connects two **Rigidbodies** together but allows the distance between them to change as though they were connected by a spring. More info

Swing Axis:

A **joint** property that defines the axis around which the **joint** can swing. More info

Swing Limit:

A **joint** property that limits the rotation around one element of the defined **Swing Axis**. More info

Target Position:

A **joint** property to set the **target position** that the **joint**'s drive force should move it to. More info

Target Velocity:

A **joint** property to set the desired velocity with which the **joint** should move to the **Target Position** under the drive force. More info

Terrain Collider:

A terrain-shaped **collider component** that handles **collisions** for **collision** surface with the same shape as the **Terrain** object it is attached to. More info

Wheel Collider:

A special **collider** for grounded vehicles. It has built-in **collision detection**, wheel physics, and a slip-based tire friction model. It can be used for objects other than wheels, but it is specifically designed for vehicles with wheels. More info

Platforms Terms

ADB:

An Android Debug Bridge (ADB). You can use an **ADB** to deploy an Android package (APK) manually after building. More info

ADT:

An Android project type that is no longer supported by Google and is considered obsolete. More info

AOT Compilation:

Ahead of Time (AOT) compilation is an **iOS** optimization method for optimizing the size of the built **iOS** player More info

APK:

The Android Package format output by Unity. An APK is automatically deployed to your device when you select File > Build & Run. More info

AR:

Augmented Reality (AR) uses computer graphics or video composited on top of a live video feed to augment the view and create interaction with real and virtual objects.

Cardboard:

See **Google Cardboard**

Gesture:

A **HoloLens** input type that uses hand signals to signify commands to the system. More info

Google Cardboard:

Google's **virtual reality** (VR) platform for smartphones. More info

Gradle:

An Android build system that automates several build processes. This automation means that many common build errors are less likely to occur. More info

Holographic:

The former name for **Windows Mixed Reality**. More info

HoloLens:

An **XR** headset for using apps made for the **Windows Mixed Reality** platform. More info

iOS:

Apple's mobile operating system. More info

Keystore:

An Android system that lets you store cryptographic key entries for enhanced device security. More info

Oculus:

A VR platform for making applications for Rift and mobile VR devices. More info

Oculus Rift:

A VR headset for using apps made for the **Oculus** platform. More info

ODR:

On-demand resources (ODR) is a feature available for the **iOS** and tvOS platforms, from version 9.0 of **iOS** and tvOS onwards. It allows you to reduce the size of your application by separating the core **Assets** (those that are needed from application startup) from **Assets** which may be optional, or which appear in later levels of your game. More info

PhotoCapture:

An API that enables you to take photos from a **HoloLens** web **camera** and store them in memory or on disk. More info

PlayStation 4, PS4:

Sony's eighth generation video **game console**.

Razor:

A CPU/GPU chip set used in **PS4** hardware. More info

Rift:

See **Oculus** Rift.

Unity Remote:

A downloadable app designed to help with Android, **iOS** and tvOS development. The app connects with Unity while you are running your project in Play Mode from the Unity Editor. More info

Virtual Reality:

A system that immerses users in an artificial 3D world of realistic images and sounds, using a headset and motion tracking. More info

VR Trace:

A **virtual reality** diagnostic tool by Sony Interactive Entertainment for PlayStation 4. More info

Wii U:

Nintendo's eighth generation video **game console**.

Windows Holographic:

The former name for **Windows Mixed Reality**. More info

Windows Mixed Reality:

A mixed reality platform developed by Microsoft, built around the API of Windows 10. More info

Xbox One:

Microsoft's eighth generation video **game console**.

XR:

An umbrella term encompassing **Virtual Reality** (VR), Augmented Reality (AR) and Mixed Reality (MR) applications. Devices supporting these forms of interactive applications can be referred to as **XR** devices. More info

Scripting Terms

Event System:

A way of sending events to objects in the application based on input, be it keyboard, mouse, touch, or custom input. The **Event System** consists of a few **components** that work together to send events. More info

IL2CPP:

A Unity-developed scripting back-end which you can use as an alternative to Mono when building projects for some platforms. More info

mcs:

The Mono C# compiler file format. More info

Mono:

A **scripting backend** used in Unity. More info

MonoDevelop:

An integrated development environment (IDE) supplied with Unity 2017.3 and previous versions. From Unity 2018.1 onwards, **MonoDevelop** is replaced by Visual Studio. More info

Scripting Backend:

A framework that powers scripting in Unity. Unity supports three different **scripting backends** depending on target platform: Mono, .NET and **IL2CPP**. **Universal Windows Platform**, however, supports only two: .NET and **IL2CPP**. More info

Scripting Event:

A way of allowing a user-driven callback to persist from edit time to run time without the need for additional programming and script configuration More info

Scripts:

A piece of code that allows you to create your own **Components**, trigger game events, modify **Component** properties over time and respond to user input in any way you like. More info

Tag:

A reference word which you can assign to one or more **GameObjects** to help you identify **GameObjects** for scripting purposes. For example, you might define and "Edible" Tag for any item the player can eat in your game. More info

Test Runner:

A Unity tool that tests your code in both Edit mode and Play mode, and also on target platforms such as Standalone, Android, or **iOS**. More info

Texture Format:

A file format for handling textures during realtime **rendering** by 3D graphics hardware, such as a graphics card or mobile device. More info

Services Terms

Client RSA Public Key:

An optional security layer that Unity can use when communicating between Unity and Xiaomi servers. Use it to receive server callbacks for client-side receipt validation, or to integrate with Unity server APIs. More info

Cloud Build:

A continuous integration service for Unity projects that automates the process of creating builds on Unity's servers. More info

Collaborate:

A Unity cloud-hosted service that provides a simple way for teams to save, share, and contribute to their Unity project. More info

Performance Reporting:

A Unity Service that captures and aggregates exception data, to help you understand what's happening during run time and to optimize your project faster. More info

Services:

A Unity facility that provides a growing range of complimentary services to help you make games and engage, retain and monetize audiences. More info

Unity Cloud Build:

See **Cloud Build** More info

Timeline Terms

animatable property:

A property belonging to a **GameObject**, or belonging to a **component** added to a **GameObject**, that can have different values over time. More info

animation:

A collection of images that create a moving image when played sequentially. In Unity, an animation is the result of adding two different animation keys, at two different times, for the same **animatable property**. More info

animation curve:

The curve drawn between keys set for the same **animatable property**, at different frames or seconds. The position of the tangents and the selected **interpolation** mode for each key determines the shape of the **animation curve**. More info

binding or Track binding:

Refers to the link between **Timeline Asset** tracks and the **GameObjects** in the **scene**. When you link a **GameObject** to a track, the track animates the **GameObject**. Bindings are stored as part of the **Timeline instance**. More info

blend area:

The area where two **Animation clips**, **Audio clips**, or Control clips overlap. The overlap creates a transition that is referred to as a blend. The duration of the overlap is referred to as the **blend area**. The **blend area** sets the duration of the transition. More info

Blend In curve:

In a blend between two **Animation clips**, **Audio clips**, or Control clips, there are two blend curves. The blend curve for the **incoming clip** is referred to as the **Blend In curve**. More info

Blend Out curve:

In a blend between two **Animation clips**, **Audio clips**, or Control clips, there are two blend curves. The blend curve for the **out-going clip** is referred to as the **Blend Out curve**. More info

Blend Type:

A Blend Tree property that allows you to configure the Blend Tree for 1D, 2D or Direct blending. More info

clip:

A generic term that refers to any clip within the **Clips view** of the **Timeline** Editor window. More info

Clips view:

The area in the **Timeline** Editor window where you add, position, and manipulate clips. More info

Control:

A function for displaying text, buttons, checkboxes, scrollbars and other features on the user interface. More info

Curves view:

The area in the **Timeline** Editor window that shows the **animation curves** for **Infinite clips** or for **Animation clips** that have been converted from **Infinite clips**. The **Curves view** is similar to Curves mode in the Animation window. More info

field:

A generic term that describes an editable box that the user clicks to enter a value. Editable fields in the **inspector** are also commonly referred to as fields. More info

Gap extrapolation:

How an Animation track approximates animation data in the gaps before and after an **Animation clip**. More info

incoming clip:

The second clip in a blend between two clips. The first clip, the **out-going clip**, transitions to the second clip, the **incoming clip**. More info

infinite clip:

A special **animation clip** that contains basic key animation recorded directly to an Animation track within the **Timeline** Editor window. An **Infinite clip** cannot be positioned, trimmed, or split because it does not have a defined duration: it spans the entirety of an Animation track. More info

interpolation:

The estimation of values that determine the shape of an **animation curve** between two keys. More info

interpolation mode:

The **interpolation** algorithm that draws the **animation curve** between two keys. The **interpolation** mode also joins or breaks left and right tangents. More info

key:

Can refer to an **Input Key** or an **Animation Key**

out-going clip:

The first clip in a blend between two clips. The first clip, the **out-going clip**, transitions to the second clip, the **incoming clip**. More info

Playhead Location field:

The field that expresses the location of the **Timeline Playhead** in either frames or seconds, depending on the **Timeline** Settings. More info

property:

A generic term for the editable fields, buttons, checkboxes, or menus that comprise a **component**. An editable property is also referred to as a field. More info

tangent:

One of two handles that controls the shape of the **animation curve** before and after a key. Tangents appear when a key is selected in the **Curves view**, or when a key is selected in the Curve Editor.

tangent mode:

The selected **interpolation** mode used by the left tangent, right tangent, or both tangents.

Timeline Asset:

Refers to the tracks, clips, and recorded animation that comprise a cinematic, cut-scene, game-play sequence, or other effect created with the **Timeline** Editor window. A **Timeline Asset** does not include bindings to the **GameObjects** animated by the **Timeline Asset**. The bindings to **scene GameObjects** are stored in the **Timeline instance**. The **Timeline Asset** is project-based. More info

Timeline Editor window:

The name of the window where you create, modify, and preview a **Timeline instance**. Modifications to a **Timeline instance** also affects the **Timeline Asset**. More info

Timeline instance:

Refers to the link between a **Timeline Asset** and the **GameObjects** that the **Timeline Asset** animates in the **scene**. You create a **Timeline instance** by associating a **Timeline Asset** to a **GameObject** through a Playable Director **component**. The **Timeline instance** is scene-based. More info

Timeline:

Generic term within Unity that refers to all features, windows, editors, and **components** related to creating, modifying, or reusing cut-scenes, cinematics, and game-play sequences. More info

Timeline Playback Controls:

The row of buttons and fields in the **Timeline** Editor window that controls playback of the **Timeline instance**. The **Timeline** Playback Controls affect the location of the **Timeline Playhead**. More info

Timeline Playback mode:

The mode that previews the **Timeline instance** in the **Timeline** Editor window. **Timeline** Playback mode is a simulation of Play mode. **Timeline** Playback mode does not support audio playback. More info

Timeline Playhead:

The white marker and line that indicates the exact point in time being previewed in the **Timeline** Editor window. More info

Timeline Selector:

The name of the menu in the **Timeline** Editor window that selects the **Timeline instance** to be previewed or modified. More info

track:

A generic term that refers to any track within the **Track list** of the **Timeline** Editor window. More info

Track group:

The term for a series of tracks organized in an expandable and collapse collection of tracks. More info

Track list:

The area in the **Timeline** Editor window where you add, group, and modify tracks. More info

UI Terms

Canvas:

The area that contains all **UI** elements in a **scene**. The Canvas area is shown as a rectangle in the **Scene View**. More info

Canvas Group:

A group of **UI** elements within a Canvas. Use a **Canvas Group** to control a group of **UI** elements collectively without needing to handle them each individually. More info

Canvas Renderer:

Renders a graphical **UI** object contained within a Canvas. More info

Canvas Scaler:

Controls the overall scale and **pixel** density of all **UI** elements in the Canvas, including font sizes and image borders. More info

Image control:

An Image control displays a non-interactive image to the user, such as a decoration and icon. You can change the image from a script to reflect changes in other controls. More info

Interactable:

A **UI component** property that detemines whether the **component** can accept input. More info

UI Mask:

A visual **component** that lets you restrict images from view to only a small section of an image. For instance, you can apply a Mask to a Panel **UI** element to block all child images from view. More info

Raw image:

A Visual **UI Component** that displays a non-interactive image to the user. This can be used for decoration, icons, **etc**, and the image can also be changed from a script to reflect changes in other controls. More info

ScrollView:

A **UI** Control which displays a large set of Controls in a viewable area that you can see by using a scrollbar. More info

Shadow:

A **UI component** that adds a simple outline effect to graphic **components** such as Text or Image. It must be on the same **GameObject** as the graphic **component**. More info

Text:

A non-interactive piece of text to the user. This can be used to provide captions or labels for other GUI controls or to display instructions or other text. More info

TextField control:

A **TextField control** displays a non-interactive piece of text to the user, such as a caption, label for other GUI controls, or instruction. More info

Text Input Field:

A field that allows the user to input a Text string More info

Toggle:

A checkbox that allows the user to switch an option on or off. More info

Toolbar:

A row of buttons and basic controls at the top of the Unity Editor that allows you to interact with the Editor in various ways (e.g. scaling, translation). More info

UI:

(User Interface) Allows a user to interact with your application. More info

Visual Component:

A **component** that enables you to easily create GUI-specific functionality. More info